NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A COMPUTER SIMULATION STUDY OF A SINGLE RIGID BODY DYNAMIC MODEL FOR BIPED POSTURAL CONTROL

by

Mehmet Bediz

March 1997

Thesis Co-Advisors: Robert B. McGhee
Michael J. Zyda

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1997	Master's Thesis

4. TITLE AND SUBTITLE
A Computer Simulation Study of a Single Rigid Body Dynamic Model For Biped Postural Control

5. FUNDING NUMBERS

6. AUTHOR(S)
Bediz, Mehmet

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Existing kinematics models for humans cannot simulate movement beyond geometric constraints. On the other hand, complex dynamics models are computationally expensive for real time computer graphics applications in Virtual Environments(VE). To be able to create a more realistic, real time, and computationally efficient human model, a simple dynamic model needs to be developed.

The approach taken in this thesis was to develop a single rigid body dynamic human model with massless legs. Instead of a Lagrangian model, which complicates the calculations exponentially as the complexity of the system increases, the Newton-Euler method was chosen to derive system differential equations. Linear state feedback was used for postural control. As part of this research, a previous realistic looking human model is further developed.

The major conclusion of this thesis is that a single rigid body dynamic model can be used for simulation of postural control. The simulation results contained in this thesis show that such a modeling technique could be used to cause a detailed kinematic representation of a human figure to move in a smooth and realistic way without resorting to complexity of a multi-link dynamic model.

14. SUBJECT TERMS	15. NUMBER OF PAGES
physically base modeling, virtual environment, articulated humans, human modeling, kinematics, dynamics, postural control	186
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

A COMPUTER SIMULATION STUDY OF A SINGLE RIGID BODY DYNAMIC MODEL FOR BIPED POSTURAL CONTROL

Mehmet Bediz
Lieutenant JG., Turkish Navy
B.S.E.E., Turkish Naval Academy, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1997

ABSTRACT

Existing kinematics models for humans cannot simulate movement beyond geometric constraints. On the other hand, complex dynamics models are computationally expensive for real time computer graphics applications in Virtual Environments(VE). To be able to create a more realistic, real time, and computationally efficient human model, a simple dynamic model needs to be developed.

The approach taken in this thesis was to develop a single rigid body dynamic human model with massless legs. Instead of a Lagrangian model, which complicates the calculations exponentially as the complexity of the system increases, the Newton-Euler method was chosen to derive system differential equations. Linear state feedback was used for postural control. As part of this research, a previous realistic looking human model is further developed.

The major conclusion of this thesis is that a single rigid body dynamic model can be used for simulation of postural control. The simulation results contained in this thesis show that such a modeling technique could be used to cause a detailed kinematic representation of a human figure to move in a smooth and realistic way without resorting to complexity of a multi-link dynamic model.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Many thanks to all whose help made this thesis possible. Special thanks to my two thesis advisors. To Dr. Michael Zyda I owe much for his guidance in Computer Graphics. My sincerest thanks to Dr. Robert McGhee for his patience, encouragement, and devotion to his students. To be able to share his vast experience made working with him a true delight. I would also like to thank to William Frey and Paul Skopowski for their support. Finally many thanks to all members of NPSNET Research Group and to all of the faculty, students and staff of the Computer Science Department who helped in numerous ways.

I. INTRODUCTION

A. MOTIVATION

There is an increasing demand for realistic virtual environments (VE). One of the main branches of research in VE is human motion simulation. Kinematics and dynamics are the two disciplines which are used to create physically based mathematical models for human motion simulation. Kinematic models are quite simple to implement and computationally inexpensive in comparison to dynamic models. However, they are limited to simulation of the geometric constraints of the body parts of the human. On the other hand, dynamic models introduce additional physical properties of the body parts to the simulation, such as mass and moment of inertia, which provides more realism. It is possible to simulate human motion realistically with detailed dynamic models. However, the cost of this realism is a high degree of computational complexity. When more detailed models are chosen, the response time of the simulated human model increases. On the other hand, reducing latency to a minimum amount of time is a important requirement for virtual environments in which humans can interact. This places limits on the complexity of the model which can be used in a particular situation.

B. GOALS

The purpose of this thesis is to develop a single rigid body dynamic human model with massless legs to illustrate that such a model can simulate human motion more smoothly and more realistically than kinematic models, and without resorting to the complexity of multi-link dynamic models. The Newton-Euler method is chosen instead of the Lagrangian method, since the complexity of the latter grows exponentially when the degrees of freedom increase.

C. ORGANIZATION

Chapter II of this thesis provides background information and reviews previous work relating to the area of kinematic and dynamic methods to simulate human motion. In this chapter, articulated rigid body dynamics, postural control, and control of stepping are also investigated in addition to single rigid body dynamics. Chapter III provides an overview of the problem statement for this thesis and discusses mathematical modeling of postural control of a single rigid body with an attached massless leg. Chapter IV introduces the developed kinematic and dynamic computer models. Chapter V presents results and conclusions about the work completed. The last chapter, Chapter VI, discusses recommendations for future enhancements and further research.

II. SURVEY OF PREVIOUS WORK

A. INTRODUCTION

Mathematical models which define human motion in detail are complex and quite nonlinear. This provides a motivation to start with simple models at a high level of abstraction. Models which ignore the dynamic properties of body parts are called *kinematic* models. There are also models which consider only the torso to have dynamic properties. More complex models introduce mass and moment of inertia for more than one body part. The more dynamic properties are added, the higher becomes the complexity and non-linearity of the resulting model.

B. KINEMATIC MODELS

Kinematics is concerned with motion without considering the forces and torques that cause it. It establishes relations between position, velocity, acceleration, and higher order derivatives of position variables.

The human body can be thought as a set of connected body parts (links). Each body part (link) has four link parameters to be defined: link length, (a), link twist (α), link offset(d) and, joint angle(θ). There exists a coordinate frame attached to each link [CRAI89].

Manipulator kinematics investigates the transformation from frame to frame as the body articulates. Forward kinematics show how to compute the position and orientation of the end-effector according to link parameters. On the other hand, inverse kinematics solves for link parameters when the position and orientation of the end-effector is given.

For the human body, it can be assumed that the joint angle is the only variable link parameter. Forward kinematics can be used to solve for the position and orientation of each body part as joint angles are given. However, this method is difficult for the animator. The

position and orientation of each body part can also be computed with the position and orientation of the end-effector as an input by using inverse kinematics. Even though the second method introduces more complexity, in this approach the animator is able to define the path for the end-effector.

C. SINGLE RIGID BODY DYNAMICS

An advanced approach to human motion simulation introduces dynamics. There are three most often used methods: free body method with hard constraints, free body method with soft constraints, and generalized coordinates with Lagrangian[MCGH79].

1. Free Body Method with Hard Constraints

This method looks at each body part as a separate free element, under the influences of joint torques, gravity, and reaction forces and moments. The simplest possible dynamical model for human motion is an inverted pendulum which idealizes the entire body as a single rigid link with upright posture maintained by ankle torque.

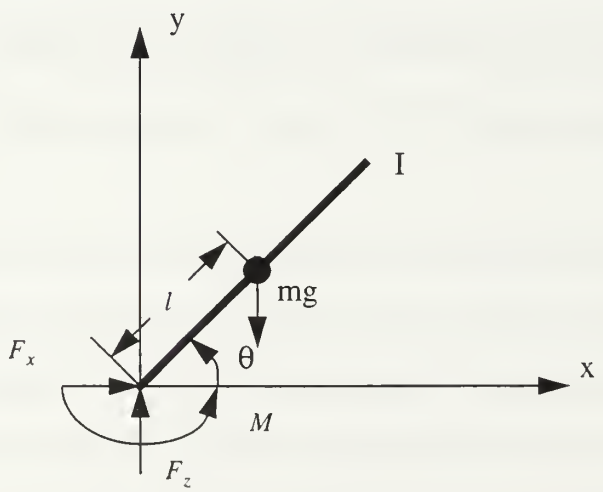

Figure 1: Inverted Pendulum

In human motion, joint torques are created by muscular contraction. The origin of reaction forces and moments is forces from other limbs or the ground. The force and torque equations for the inverted pendulum are [MCGH79]

$$m\ddot{x} = F_x \qquad \text{(eq. 2.1)}$$

$$m\ddot{y} = F_y - mg \qquad \text{(eq. 2.2)}$$

$$I\ddot{\theta} = F_x l\sin\theta - F_y l\cos\theta + M \qquad \text{(eq. 2.3)}$$

where l is the distance from the coordinate origin to the center of gravity, I is moment of inertia about the center of gravity, F_x, F_y are ground reaction forces, and M is the joint control torque which is computed by some control law

$$M = u(x, t). \qquad \text{(eq. 2.4)}$$

In this equation, x is the system *state vector* and t is time.

After expressing reaction forces and moments, constraint equations are derived according to the geometry of the system, and differentiated twice. Constraint equations for this example are

$$x = l\cos\theta \qquad \text{(eq. 2.5)}$$

and

$$y = l\sin\theta \qquad \text{(eq. 2.6)}$$

with second derivatives

$$\ddot{x} = -\ddot{\theta}l\sin\theta - \dot{\theta}^2 l\cos\theta \qquad \text{(eq. 2.7)}$$

and

$$\ddot{y} = \ddot{\theta}l\cos\theta - \dot{\theta}^2 l\sin\theta \qquad \text{(eq. 2.8)}$$

The five equations above can be solved numerically or analytically to determine the unknowns: $\ddot{x}$, $\ddot{y}$, $\ddot{\theta}$, F_x, F_y. The matrix form of the equations is

$$\begin{bmatrix} m & 0 & 0 & -1 & 0 \\ 0 & m & 0 & 0 & -1 \\ 0 & 0 & I & -l\sin\theta & l\cos\theta \\ 1 & 0 & l\sin\theta & 0 & 0 \\ 0 & 1 & -l\cos\theta & 0 & 0 \end{bmatrix} \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{\theta} \\ F_x \\ F_y \end{bmatrix} = \begin{bmatrix} 0 \\ -mg \\ M \\ -\dot{\theta}^2 l\cos\theta \\ -\dot{\theta}^2 l\sin\theta \end{bmatrix} \qquad \text{(eq. 2.9)}$$

The behavior of the pendulum can also be expressed with a 3x3 matrix multiplication by analytically eliminating $\ddot{x}$ and $\ddot{y}$. The 3x3 matrix equation is [MCGH79b]

$$\begin{bmatrix} I & -l\sin\theta & l\cos\theta \\ ml\sin\theta & 1 & 0 \\ -ml\cos\theta & 0 & 1 \end{bmatrix} \begin{bmatrix} \ddot{\theta} \\ F_x \\ F_y \end{bmatrix} = \begin{bmatrix} M \\ -m\dot{\theta}^2 l\cos\theta \\ -m\dot{\theta}^2 l\sin\theta + mg \end{bmatrix} \qquad \text{(eq. 2.10)}$$

Since this system has only a single degree of freedom, a suitable state vector is

$$x = \begin{pmatrix} \theta \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \qquad \text{(eq. 2.11)}$$

so

$$\dot{x} = \begin{pmatrix} \dot{\theta} \\ \ddot{\theta} \end{pmatrix} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} \qquad \text{(eq. 2.12)}$$

For simulating this model, numerical integration can be used to compute $x(t)$ for any given $x(0)$. The value of $\ddot{\theta}$ (or $\dot{x}_2$) is calculated by the system dynamic equations. It

has been found in the work of this thesis that the 3x3 matrix solution is approximately two times faster than the 5x5 one.

2. Free Body Method with Soft Constraints

A more realistic approach is to consider the connection between limb segments as not rigid. In the *soft constraint* method, the constraint equations are replaced with functions defined by joint variables and their derivatives, which are used to compute constraint forces with some gain values. This increases the order of state equation by a factor of three in the planar case, and by a factor of six in the three dimensional case, but avoids matrix inversion.

The previous inverted pendulum model becomes as shown in Figure 2 for the soft constraint method.

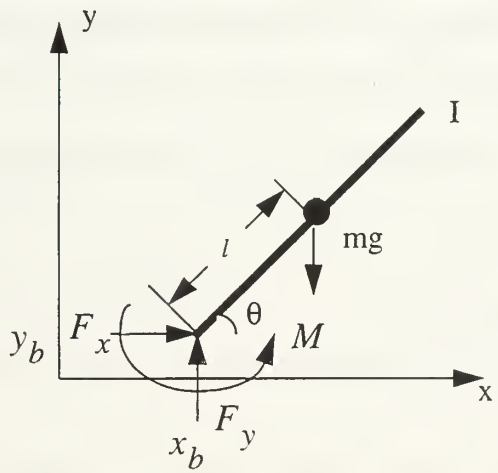

Figure 2: Inverted Pendulum with Soft Constrains

Instead of Eq. 2.5 and Eq. 2.6, the soft constraint equations are[MCGH79]

$$x = x_b + l\cos\theta \qquad \text{(eq. 2.13)}$$

$$y = y_b + l\sin\theta \qquad \text{(eq. 2.14)}$$

with the corresponding soft constraint forces

$$F_x = -k_x x_b - k_{\dot{x}} \dot{x}_b \qquad \text{(eq. 2.15)}$$

$$F_y = -k_y y_b - k_{\dot{y}} \dot{y}_b \qquad \text{(eq. 2.16)}$$

The state vector for the numerical solution is

$$x = (\theta, \dot{\theta}, x_b, \dot{x_b}, y_b, \dot{y_b})^T \qquad \text{(eq. 2.17)}$$

After obtaining the constraint forces by using the state vector values, Eq 2.1, Eq 2.2, and Eq 2.3 can be solved directly for translational and angular accelerations instead of requiring matrix inversion.

3. Generalized Coordinates with Lagrangian

Another way to derive the system equations is to use the Lagrangian method. This method does not compute constraint forces and moments, but instead uses the difference of kinetic and potential energies expressed in terms of *generalized coordinates*.

The *virtual work* function, for any generalized coordinate, θ, is

$$\delta W = Q_\theta \delta\theta \qquad \text{(eq. 2.18)}$$

The Lagrangian function is

$$L = K - V \qquad \text{(eq. 2.19)}$$

where K is the kinetic energy function and V is the potential energy function. The differential equations of the system are obtained from [MCGH79b]

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = Q_{q_i} \qquad \text{(eq. 2.20)}$$

where q_i represents the generalized coordinate for each i and Q_{q_i} is the coefficient of δq_i in the virtual work function.

Considering the hard constraint example, Figure 1, the virtual work function is

$$\delta W = M\delta\theta \qquad \text{(eq. 2.21)}$$

where θ is the generalized coordinate. The kinetic energy of this system is

$$T = \frac{1}{2}(m\dot{x}^2 + m\dot{y}^2 + I\dot{\theta}^2) \qquad \text{(eq. 2.22)}$$

or

$$T = \frac{1}{2}(ml^2\dot{\theta}^2\sin\theta + ml^2\dot{\theta}^2\cos\theta + I\dot{\theta}^2) \qquad \text{(eq. 2.23)}$$

which simplifies to

$$T = \frac{1}{2}(ml^2 + I)\dot{\theta}^2 \qquad \text{(eq. 2.24)}$$

The potential energy of the system is

$$V = mgl\sin\theta \qquad \text{(eq. 2.25)}$$

Thus, the Lagrangian function is

$$L = \frac{1}{2}(ml^2 + I)\dot{\theta}^2 - mgl\sin\theta \qquad \text{(eq. 2.26)}$$

By evaluation the Lagrangian derivatives, the result is that the angular acceleration of the pendulum is given by

$$\ddot{\theta} = \frac{M - mgl\cos\theta}{I + ml^2} \qquad \text{(eq. 2.27)}$$

For a numerical solution, the state vector is

$$x = \begin{pmatrix}\theta \\ \dot{\theta}\end{pmatrix} = \begin{pmatrix}x_1 \\ x_2\end{pmatrix} \qquad \text{(eq. 2.28)}$$

The Lagrangian approach is perhaps the most effective one for simple cases. However, with an increase in the number of degrees of freedom of the system the complexity of the calculations grows exponentially.

D. ARTICULATED RIGID BODY DYNAMICS

It is certainly impossible to model the human body with complete realism as a single mass rigid body. One can, however, abstract it as a system of rigid bodies connected together with rotary joints with control torques acting at each joint. If the motion of each limb segment is given, and the forces and torques are to be computed, this problem is called the *inverse* dynamics problem. If the torques are given and the accelerations are to be determined, this is called the *direct* dynamics problem [KOOZ83].

There is an algorithm, called Articulated Body(AB), for direct dynamics which contains three O(n) recursions [MCMI95]. In the first step of this method, which is Forward Kinematics, velocity and velocity dependent terms are computed from the base to the tip of each serial chain of links. The orientation of i's coordinate system with respect to $i-1$'s specified by the *rotation matrix* [CRAI89], ${}^{i}R_{i-1}$, can be determined by using joint position, q_i, which is an element of state vector. The position of i's coordinate system origin according to $i-1$, ${}^{i-1}p_i$, is a constant. Then the *spatial transformation* matrix is [CRAI89]

$$ {}^{i+1}X_i = \begin{bmatrix} {}^{i+1}R_i & 0 \\ {}^{i}p_{i+1} \times {}^{i+1}R_i & {}^{i+1}R_i \end{bmatrix} \qquad \text{(eq. 2.29)} $$

The spatial transformation matrices, along with $\dot{q}_i$ are used to compute velocity values of the links by using the relation

$$v_i = {}^{i}X^{T}_{i-1}\ v_{i-1} \qquad \text{(eq. 2.30)}$$

The velocity-dependent bias forces, β_i, and the vector of Coriolis and centripetal accelerations, ζ_i, of each link are also determined from the base to the tip. Details of this calculation can be found in [MCMI95b].

In the second step, which is backwards dynamics, $I^{\dagger}{}_i$ and $\beta^{\dagger}{}_i$ are computed from the tip to the base. The matrix $I^{\dagger}{}_i$ is the 6X6 inertia of links i though N. In other words, "the inertia 'felt' at the i coordinate system when the joints from $i+1$ to N are free to move"[MCMI95]. The vector $\beta^{\dagger}{}_i$ is the bias force exerted on ith link, including all outboard torques.

In the final step, joint and link accelerations are computed from the base to the tip as $a_0 = 0$ is given and by using the equations

$$a'_i = {}^{i}X_{i-1}a'_{i-1} + \phi_i\ddot{q}_i + \zeta_i \qquad \text{(eq. 2.31)}$$

and

$$f_i = I^{\dagger}{}_i a'_i - \beta^{\dagger}{}_i \qquad \text{(eq. 2.32)}$$

where f_i is the spatial force exerted onto link i by its inboard link which contains the effect of input torque, T_i, and where a' is a six element vector containing the angular

acceleration, $\dot{\omega}_i$, and translational acceleration vectors. Each ϕ_i is a six element unit vector which specifies the corresponding joint axis.

Another numerical approach to the recursive direct dynamics is based on solving the inverse dynamics problem for a sequence of n + 1 specialized acceleration vectors. In general, the inverse dynamic equation can be written as [KOOZ83]

$$T = C\ddot{\theta} - D \qquad \text{(eq. 2.33)}$$

where the matrix C is given by

$$C = B^{-1}J = [C_1 C_2 C_3 \ldots] \qquad \text{(eq. 2.34)}$$

and where each element of vector C is a column vector. Usually, C is not known explicitly. However it can be computed with a numerical approach using an inverse dynamics model such as that in [CRAI89]. Specifically, for the case $\ddot{\theta} = (0, 0, \ldots)^T$, from Eq 2.33

$$T_0 = -D \qquad \text{(eq. 2.35)}$$

The torque T_0 can be computed by using the kinematic and dynamic equations of the articulated mechanism. As an example, the model given in [KOOZ83] which is illustrated in Figure 3 can be used.

The kinematic equations of the planar model in Figure 3 are

$$\ddot{x}_i = \ddot{x}_{i-1} - (l_{i-1} - d_{i-1})(\ddot{\theta}_{i-1}\sin\theta_{i-1} + \dot{\theta}_{i-1}^2\cos\theta_{i-1})$$

$$-d_i(\ddot{\theta}_i\sin\theta_i + \dot{\theta}_i^2\cos\theta_i) \qquad \text{(eq. 2.36)}$$

and

$$\ddot{y}_i = \ddot{y}_{i-1} + (l_{i-1} - d_{i-1})(\ddot{\theta}_{i-1}\cos\theta_{i-1} - \dot{\theta}_{i-1}^2 \sin\theta_{i-1})$$

$$+ d_i(\ddot{\theta}_i \cos\theta_i - \dot{\theta}_i^2 \cos\theta_i) \qquad \text{(eq. 2.37)}$$

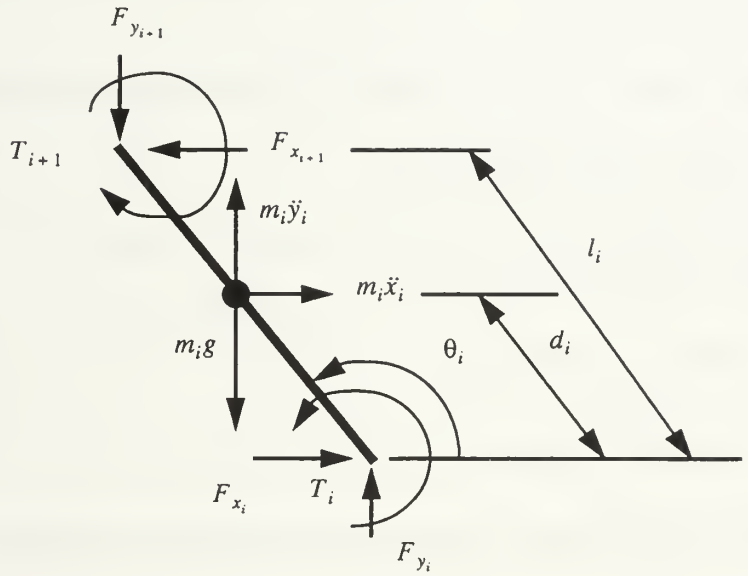

Figure 3: Free-body for Typical Link of Serial Open-chain Planar Articulated Mechanism [KOOZ83]

Eq. 2.36 and Eq. 2.37 are used to calculate the translational accelerations of the links from the base to the tip. By using these accelerations, joint torques can be computed in the same recursive manner, but from out to inward with the following equations:

$$F_{x_i} = F_{x_{i+1}} + m_i\ddot{x}_i \qquad \text{(eq. 2.38)}$$

$$F_{y_i} = F_{y_{i+1}} + m_i\ddot{y}_i + m_ig \qquad \text{(eq. 2.39)}$$

$$T_i = T_{i+1} - F_{x_{i+1}}(l_i - d_i)\sin\theta_i + F_{y_{i+1}}(l_i - d_i)\cos\theta_i \qquad \text{(eq. 2.40)}$$

$$-F_{x_i}d_i\sin\theta_i + F_{y_i}d_i\cos\theta_i + J_i\theta_i \qquad \text{(eq. 2.41)}$$

For the case $\ddot{\theta} = (1, 0, \ldots)^T$, all accelerations except $\ddot{\theta}_1$ would be equal to zero which means

$$T_1 = C_1 - D \qquad \text{(eq. 2.42)}$$

The joint torque, T_1, can be calculated with the recursive method, which is described above. Then C_1 can be calculated as

$$C_1 = T_1 + D = T_1 - T_0 \qquad \text{(eq. 2.43)}$$

With the same logic C_i is

$$C_i = T_i - T_0 \qquad \text{(eq. 2.44)}$$

Thus, the C matrix can be computed numerically by following these steps n times for a n-link system. Then $\ddot{\theta}$ is given by

$$\ddot{\theta} = C^{-1}(T - T_0) \qquad \text{(eq. 2.45)}$$

where T is any arbitrary torque and T_0 is the "equilibrium" torque defined by Eq 2.35. Because of the required matrix inversion, the second method has O(n^3) complexity. It has been shown [MCMI94], that O(n^3) methods are best for simple serial chains of rigid bodies when $n \leq 3$, and O(n) methods are better for $n > 3$.

E. POSTURAL CONTROL

It is clearly necessary to begin with modeling stable standing of a human before considering control of walking. All the methods derived in the previous sections show how to compute joint angle accelerations according to joint control torques. The question is how

to formulate control torques. One straightforward way is *linear state feedback control* [MCGH86].

A suitable state vector for postural control can be defined as

$$x = \begin{bmatrix} \theta \\ \dots \\ \dot{\theta} \end{bmatrix} \qquad \text{(eq. 2.46)}$$

where θ is an nx1 vector of joint positions and $\dot{\theta}$ is the nx1 vector of joint velocities. The corresponding body state equation is[KOOZ83]

$$\dot{x} = f(x) + E(x)\ T \qquad \text{(eq. 2.47)}$$

In Eq. 2.10 f and, E are in general not available analytically. However, with the numerical approach, explained in the previous section, the same equation can be expressed as

$$\dot{x} = \begin{bmatrix} 0 & I \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ C^{-1} \end{bmatrix} (T - T_0) \qquad \text{(eq. 2.48)}$$

where I is the unit matrix.

Linear feedback systems permit much flexibility to the designer, such as pole assignment [KOOZ83]. Since C^{-1} and T_0 are functions of x, the system is not linear. However, it can be linearized around 0 for joint angular rates and 90 degrees for joint angles for erect body position. Then the linearized body state equation can be written as

$$\dot{x} = Fx + GT \qquad \text{(eq. 2.49)}$$

where F is the linearized system matrix and G is the *control distribution* matrix [KOOZ83].

Under the assumption of linear state feedback, the control torques vector will be

$$T = K x \qquad \text{(eq. 2.50)}$$

where K is the gain matrix. If Eq. 2.12 and Eq. 2.13 are combined, the derivative of the state vector is then

$$\dot{x} = Fx + GKx = [F + GK]x = Hx \qquad \text{(eq. 2.51)}$$

Gain values in K should be chosen so that all eigenvalues of H have negative real parts in order to obtain a stable postural control [KOOZ83] [CAMA77].

F. CONTROL OF STEPPING

If all the postural control system eigenvalues have negative real parts, the articulated mechanism can maintain its upright standing posture. The next step is to add stepping to the system. At this point there are two options to choose: static or dynamic balancing. "A statically balanced system avoids tipping and ensuing horizontal accelerations by keeping the center of mass of the body over the polygon of the support, formed by the feet." [RABI86]. On the other hand, a dynamically balanced system can depart from static equilibrium and is permitted to tip and accelerate for short period of time. By observing human walking, one can easily say that dynamical balancing needs to be chosen for simulating a stepping human.

A step length control method with dynamic stability was presented by Gubina [GUBI74]. Gubina's model has a single rigid body with two supporting massless legs. The system has three degrees of freedom. The state vector for the linearized system is

$$x = (r - r_0, \dot{r}, \theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2)^T \qquad \text{(eq. 2.52)}$$

where θ_1 is the leg angle, θ_2 is the body attitude, r is the leg length, and r_0 is the desired leg length.

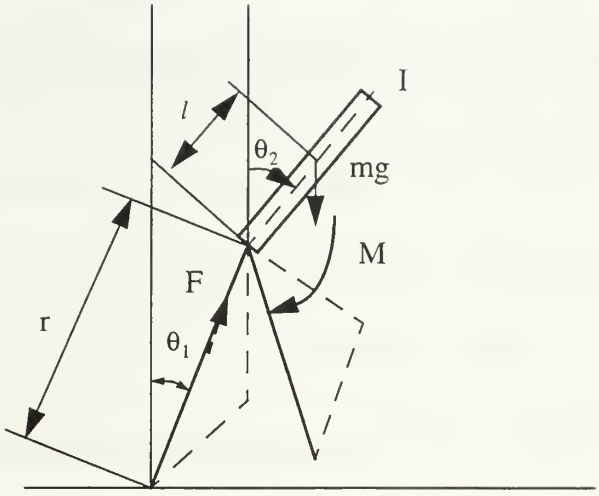

Figure 4: Single Rigid Body Model with Variable Length Massless Supporting Legs [GUBI74]

The input vector for the linear state equations is

$$u = (u_1, u_2)^T \qquad \text{(eq. 2.53)}$$

where

$$u_1 = F - F_0 \qquad \text{(eq. 2.54)}$$

and

$$u_2 = M - M_0 \qquad \text{(eq. 2.55)}$$

In these equations, F is the control force applied along the leg, M is the control torque applied at the hip and F_0, M_0 are bias force and torque respectively.

The input vector, u, is used to control leg length, and body attitude and step control are used to produce the desired forward motion. The leg length is controlled by u_1 as

$$u_1 = h_1(r - r_0) + h_2\dot{r} \qquad \text{(eq. 2.56)}$$

where h_1 and h_2 are gain constants. The body attitude is controlled by u_2 as

$$u_2 = h_5(\theta_2 - \alpha) + h_6\dot{\theta}_2 \qquad \text{(eq. 2.57)}$$

where h_5 and h_6 are gain constants and α is the bias term to permit the desired attitude.

When the initial conditions are assumed as $r(0) = r_0$, $\dot{r}(0) = 0$, $\theta_2(0) = 0$, $\dot{\theta}_2(0) = 0$, the leg angle is governed by the linearized system equation

$$\ddot{\theta}_1 - b^2\theta_1 = 0 \qquad \text{(eq. 2.58)}$$

where

$$b^2 = \frac{g}{(r_0 - l)} \qquad \text{(eq. 2.59)}$$

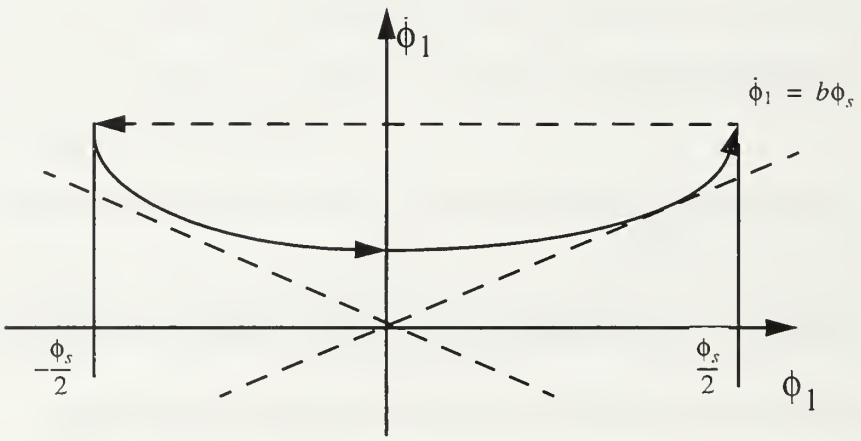

Figure 5: Stable Gait Function [GUBI74]

As seen in Figure 5, the leg has a negative angular velocity growth rate for negative angles and positive angular velocity growth rate for positive angles. The horizontal dotted line which is extended from the right to the left represents the change of supporting leg. In

the absence of disturbances, the leg angle can be described with a periodic function, $\theta_1(t)$. The existence of such a function provides a "stable gait". However, disturbances do occur in locomotion systems. That is why feedback control is needed. With such a control mechanism, the leg angle differential equation becomes

$$\dot{x}_s = \begin{bmatrix} 0 & 1 \\ b^2 & 0 \end{bmatrix} x_s - \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_s(k)\delta(t - kT) \qquad \text{(eq. 2.60)}$$

where δ denotes the unit inpulse function and

$$x_s = (x_3, x_4)^T \qquad \text{(eq. 2.61)}$$

It is shown in [GUBI74] that a suitable discrete time control vector for this system is

$$u_s(k) = h_3[x_3(kT) - \theta_s] + h_4\left[x_4(kT) - \frac{v_0}{r_0}\right] + \theta_0 \qquad \text{(eq. 2.62)}$$

The angle, θ_s, is the desired total leg angle excursion over one cycle, while v_0 is the desired forward speed.

Another foot placement algorithm for controlling forward speed is investigated by Raibert[RABI74]. In this work, a one legged hopping machine was developed to investigate dynamically balanced running robots. The machine has two main parts: the body and the leg which is connected to the body with a hinge. There also exists a hip actuator to be able to apply a torque from the body to the leg.

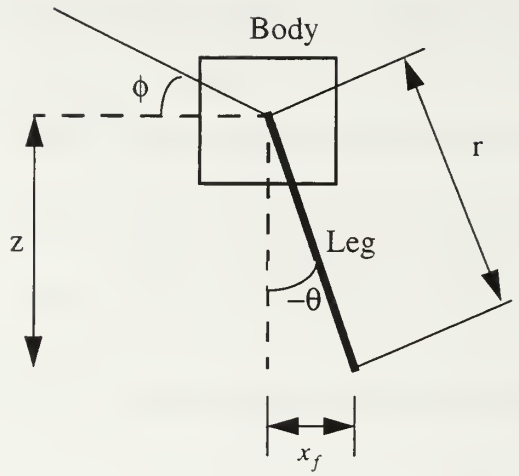

Figure 6: One Legged Hopping Machine with Control Variables [RAIB86]

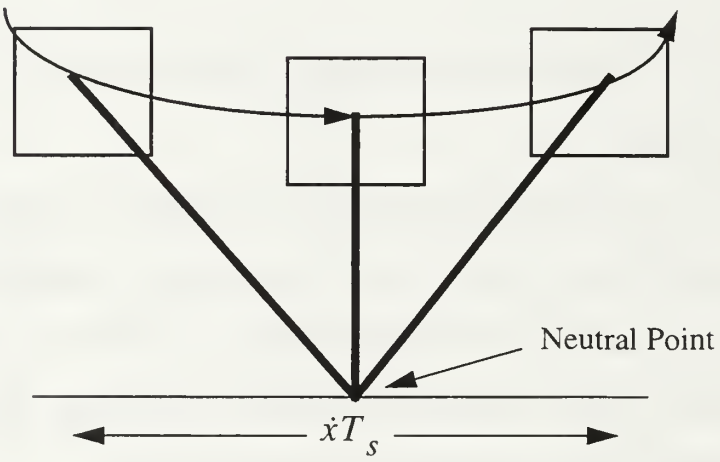

Figure 7: Placing The Foot at The Neutral Point [RABI74]

For each forward speed there is a foot position which causes zero acceleration in the direction of motion. This point is called the "neutral point". Placing the foot behind the neutral point accelerates the machine, placing the foot in front of the neutral point decelerates it.

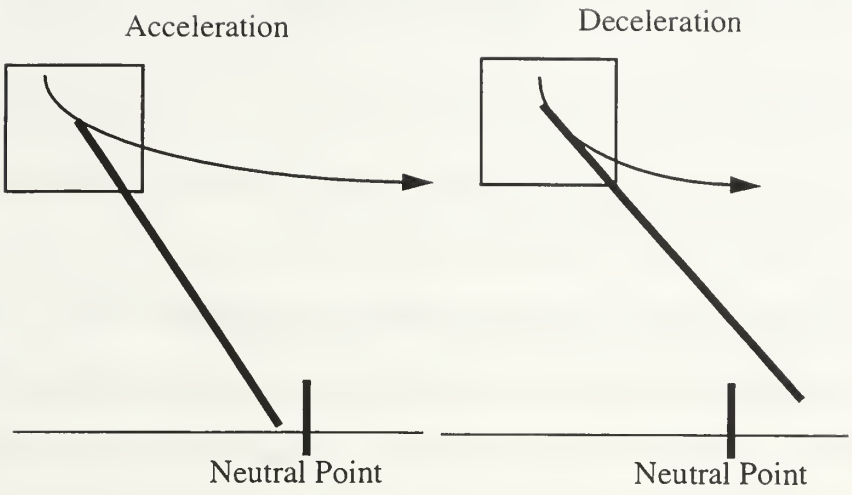

Figure 8: Acceleration and Deceleration of the one Legged Hopping Machine According to Neutral Point[RAIB86]

To place the foot at the neutral point, the control system should extend the leg such that [RAIB86]

$$x_{f_0} = \frac{\dot{x}T_s}{2} \qquad \text{(eq. 2.63)}$$

where x_{f_0} is the forward displacement of the foot with respect to the center mass, $\dot{x}$ is the forward speed, and T_s is the duration of the stance phase. An additional displacement is needed to correct the errors in the forward speed:

$$x_{f_\Delta} = k_{\dot{x}}(\dot{x} - \dot{x}_d) \qquad \text{(eq. 2.64)}$$

where x_{f_Δ} is the displacement of the foot from the neutral point, $\dot{x}_d$ is the desired forward speed and $k_{\dot{x}}$ is the feedback gain constant.

By combining the equation Eq. 2.64 and Eq. 2.65, the total foot displacement is

$$x_f = \frac{\dot{x}T_s}{2} + k_{\dot{x}}(\dot{x} - \dot{x}_d) \qquad \text{(eq. 2.65)}$$

Then the required hip angle for the desired velocity is

$$\gamma_d = \Phi - arc\sin\left(\frac{\dot{x}T_s}{2r} + \frac{k_{\dot{x}}(\dot{x} - \dot{x}_d)}{r}\right) \qquad \text{(eq. 2.66)}$$

where γ is the angle between the leg and the body [RAIB86].

Another research focused on biped systems was conducted by Troy [TROY96]. He investigated locomotion of dynamically balanced biped mechanisms. Different multilink planar and spatial biped models were developed by using feedback control for balancing and walking. In general, his work extends the results described above while confining locomotion to a specified sagittal plane.

G. SUMMARY

This chapter provides a survey of previous work relating to modeling and control of posture and gait for a walking human. It starts with kinematic modeling and continues with single and articulated rigid body dynamic models. One section discusses simulation of postural control for erect posture of the body. The last part of the chapter illustrates two different stepping control approaches for a given desired forward speed. The next chapter of this thesis derives the differential equations of a human model which has two degrees of freedom by using the Newton-Euler method. It also discusses postural control for the same model.

III. DETAILED PROBLEM STATEMENT AND MATHEMATICAL FORMULATION

A. INTRODUCTION

The model in [GUBI74], Figure 4, is designed to simulate postural and gait control with three degrees of freedom. The body attitude and the ankle angle are used for postural control and the variable leg length and foot placement as well as periodic stepping are used for gait control. The model has one input control torque and one input control force. Another similar model, dealing only with postural control, will be introduced in this chapter. This model has two degrees of freedom with the same variable angles of the previous model. Instead of the input control force, this model, shown in Figure 9, has a second input control torque at the ankle.

B. DESCRIPTION OF THE MODEL

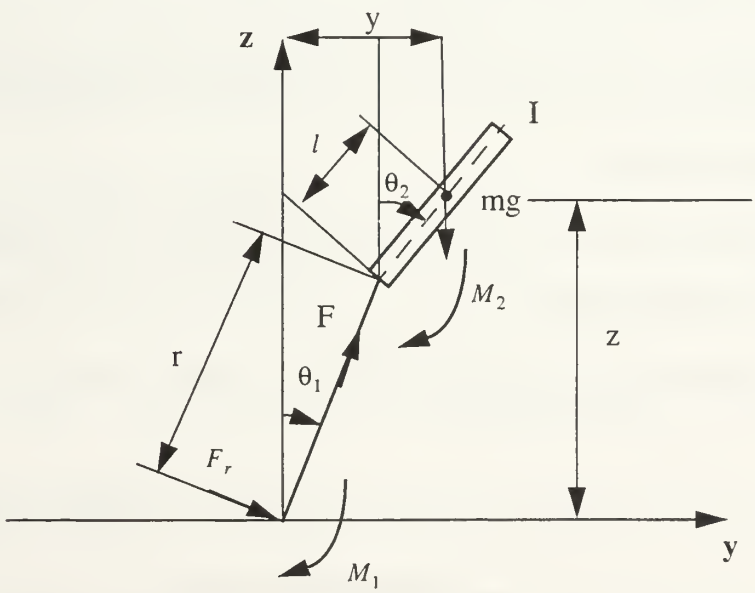

Figure 9: Single Rigid Body Model with A Constant Length Massless Supporting Leg

This model, Figure 9, has a single rigid body with one supporting massless leg. The system has two degrees of freedom. The state vector for the system is

$$x = (\theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2)^T \qquad \text{(eq. 3.1)}$$

where θ_1 is the leg angle and θ_2 is the body attitude. The leg length, r, is constant.

The input vector for the state equations is

$$u = (u_1, u_2)^T \qquad \text{(eq. 3.2)}$$

where

$$u_1 = M_1 \qquad \text{(eq. 3.3)}$$

and

$$u_2 = M_2 \qquad \text{(eq. 3.4)}$$

In these equations, M_1 is the control torque applied at the ankle and M_2 is the control torque applied at the hip.

The input vector, u, is used to control leg angle and body attitude. According to general linear feedback control

$$u = \begin{pmatrix} M_1 \\ M_2 \end{pmatrix} = K(x - x_0) \qquad \text{(eq. 3.5)}$$

where x_0 is the desired state vector. A special form of this equation, called *local* feedback, has been proposed for postural control [KOOZ83][CAMA77]. In local feedback, each joint torque is determined by angle and rotation rate of that joint only. This decouples control of

joints from each other. Specifically, in this form of control, the leg angle is controlled by u_1 as

$$u_1 = k_{\theta_1}(\theta_1 - \alpha_1) + k_{\dot{\theta}_1}\dot{\theta}_1 \qquad \text{(eq. 3.6)}$$

where k_{θ_1} and $k_{\dot{\theta}_1}$ are gain constants and α_1 is the desired leg angle. The body attitude is controlled by u_2 as

$$u_2 = k_{\theta_2}(\theta_2 - \alpha_2) + k_{\dot{\theta}_2}\dot{\theta}_2 \qquad \text{(eq. 3.7)}$$

where k_{θ_2} and $k_{\dot{\theta}_2}$ are gain constants and α_2 is the bias term to permit the desired body attitude. Local control will be examined further in this thesis.

C. THE LAGRANGIAN VERSION OF THE PROBLEM

The components of the linear velocity vector of the center of mass of the single rigid body can be derived from angular variables as follows:

$$\dot{y} = r\dot{\theta}_1 \cos\theta_1 + l\dot{\theta}_2 \cos\theta_2 \qquad \text{(eq. 3.8)}$$

$$\dot{z} = -r\dot{\theta}_1 \sin\theta_1 - l\dot{\theta}_2 \sin\theta_2 \qquad \text{(eq. 3.9)}$$

The kinetic energy of the system is thus

$$K = \frac{1}{2}m(r\dot{\theta}_1 \cos\theta_1 + l\dot{\theta}_2 \cos\theta_2)^2 + \frac{1}{2}m(-r\dot{\theta}_1 \sin\theta_1 - l\dot{\theta}_2 \sin\theta_2)^2 + \frac{1}{2}I\dot{\theta}_2^2 \qquad \text{(eq. 3.10)}$$

The potential energy of the system can be expressed as

$$V = mg(r\cos\theta_1 + l\cos\theta_2) \qquad \text{(eq. 3.11)}$$

Thus, the Lagrangian function becomes

$$L = \frac{1}{2}m(r\dot{\theta}_1\cos\theta_1 + l\dot{\theta}_2\cos\theta_2)^2 + \frac{1}{2}m(-r\dot{\theta}_1\sin\theta_1 - l\dot{\theta}_2\sin\theta_2)^2$$

$$+ \frac{1}{2}I\dot{\theta}_2^2 - mg(r\cos\theta_1 + l\cos\theta_2) \qquad \text{(eq. 3.12)}$$

The differential equation for the first generalized coordinate, θ_1, is

$$\frac{d}{dt}\frac{\partial L}{\partial\dot{\theta}_1} - \frac{\partial L}{\partial\theta_1} = M_1 - M_2 \qquad \text{(eq. 3.13)}$$

The term, $\frac{\partial L}{\partial\dot{\theta}_1}$, in equation Eq. 3.12 can be derived as

$$\frac{\partial L}{\partial\dot{\theta}_1} = mr^2\dot{\theta}_1(\cos\theta_1)^2 + lmr\dot{\theta}_2\cos\theta_1\cos\theta_2$$

$$+ mr\ \dot{\theta}_1(\sin\theta_1)^2 + lmr\dot{\theta}_2\sin\theta_1\sin\theta_2 \qquad \text{(eq. 3.14)}$$

Then, the first term in the equation Eq. 3.12 is

$$\frac{d}{dt}\frac{\partial L}{\partial\dot{\theta}_1} = mr^2\ddot{\theta}_1(\cos\theta_1)^2 - 2mr^2\dot{\theta}_1^2\cos\theta_1\sin\theta_1$$

$$+ lmr\ddot{\theta}_2\cos\theta_1\cos\theta_2 - lmr\dot{\theta}_2\dot{\theta}_1\sin\theta_1\cos\theta_2$$

$$- lmr\dot{\theta}_2^2\cos\theta_1\sin\theta_2 + mr^2\ddot{\theta}_1(\sin\theta)^2$$

$$+ 2mr^2\dot{\theta}_1^2\sin\theta_1\cos\theta_2 + mlr\ddot{\theta}_2\sin\theta_1\sin\theta_2$$

$$+ mlr\dot{\theta}_2\dot{\theta}_1\cos\theta_1\sin\theta_2 + mlr\dot{\theta}_2^2\sin\theta_1\cos\theta_2 \qquad \text{(eq. 3.15)}$$

And the second term in the equation Eq. 3.12 is

$$\frac{\partial L}{\partial \theta_1} = -mr^2\dot{\theta}_1^2 \sin\theta_1 \cos\theta_1 - mrl\dot{\theta}_1\dot{\theta}_2 \sin\theta_1 \cos\theta_2$$

$$+ mr^2\dot{\theta}_1^2 \cos\theta_1 \sin\theta_1 + mrl\dot{\theta}_1\dot{\theta}_2 \cos\theta_1 \sin\theta_2$$

$$+ mgr \sin\theta_1 \qquad \text{(eq. 3.16)}$$

By substituting Eq. 3.14 and Eq. 3.15 into Eq. 3.12, the following equation can be obtained:

$$mr^2\ddot{\theta}_1 + lmr\ddot{\theta}_2(\sin\theta_1 \sin\theta_2 + \cos\theta_1 \cos\theta_2)$$

$$+ lmr\dot{\theta}_2^2(\sin\theta_1 \cos\theta_2 - \cos\theta_1 \sin\theta_2)$$

$$+ lmr\dot{\theta}_2\dot{\theta}_1(\cos\theta_1 \sin\theta_2 - \sin\theta_1 \cos\theta_2)$$

$$+ lmr\dot{\theta}_1\dot{\theta}_2(\sin\theta_1 \cos\theta_2 - \cos\theta_1 \sin\theta_2)$$

$$-mgr \sin\theta_1 = M_1 - M_2 \qquad \text{(eq. 3.17)}$$

After applying trigonometric conversion rules, the first differential equation of the model can be derived as follows:

$$mr^2\ddot{\theta}_1 + lmr\ddot{\theta}_2 \cos(\theta_2 - \theta_1) + lmr\dot{\theta}_2^2 \sin(\theta_1 - \theta_2)$$

$$-mgr \sin\theta_1 = M_1 - M_2 \qquad \text{(eq. 3.18)}$$

The differential equation for the second generalized coordinate, θ_2, is

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}_2} - \frac{\partial L}{\partial \theta_2} = M_2 \qquad \text{(eq. 3.19)}$$

The term, $\frac{\partial L}{\partial \dot{\theta}_2}$, equation in Eq. 3.19 can be derived as

$$\frac{\partial L}{\partial \dot{\theta}_2} = mlr\dot{\theta}_1 \cos\theta_1 \cos\theta_2 + ml^2\dot{\theta}_2(\cos\theta_2)^2$$

$$+ mlr\dot{\theta}_1 \sin\theta_1 \sin\theta_2 + ml^2\dot{\theta}_2(\sin\theta_2)^2 + I\dot{\theta}_2 \qquad \text{(eq. 3.20)}$$

The time derivative of Eq. 3.20 is

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}_2} = mlr\ddot{\theta}_1 \cos\theta_1 \cos\theta_2 - mlr\dot{\theta}_1^2 \sin\theta_1 \cos\theta_2$$

$$- mlr\dot{\theta}_1\dot{\theta}_2 \cos\theta_1 \sin\theta_2 + ml^2\ddot{\theta}_2(\cos\theta_2)^2$$

$$- 2ml^2\dot{\theta}_2^2 \cos\theta_2 \sin\theta_2 + mlr\ddot{\theta}_1 \sin\theta_1 \sin\theta_2$$

$$+ mlr\dot{\theta}_1^2 \cos\theta_1 \sin\theta_2 + mlr\dot{\theta}_1\dot{\theta}_2 \sin\theta_1 \cos\theta_2$$

$$+ ml^2\ddot{\theta}_2(\sin\theta_2)^2 + 2ml^2\dot{\theta}_2^2 \sin\theta_2 \cos\theta_2 + I\ddot{\theta}_2 \qquad \text{(eq. 3.21)}$$

Then the second term in the equation Eq. 3.18 can be derived as

$$\frac{\partial L}{\partial \theta_2} = -mlr\dot{\theta}_1\dot{\theta}_2 \cos\theta_1 \sin\theta_2 + mlr\dot{\theta}_1\dot{\theta}_2 \sin\theta_1 \cos\theta_2$$

$$+ mgl\sin\theta_2 \qquad \text{(eq. 3.22)}$$

By substituting Eq. 3.21 and Eq. 3.22 into Eq. 3.19, the following equation can be obtained:

$$mlr\ddot{\theta}_1 \cos\theta_1 \cos\theta_2 - mlr\dot{\theta}_1^2 \sin\theta_1 \cos\theta_2$$

$$- mlr\dot{\theta}_1\dot{\theta}_2 \cos\theta_1 \sin\theta_2 + ml^2\ddot{\theta}_2(\cos\theta_2)^2$$

$$- 2ml^2\dot{\theta}_2^2 \cos\theta_2 \sin\theta_2 + mlr\ddot{\theta}_1 \sin\theta_1 \sin\theta_2$$

$$+ mlr\dot{\theta}_1^2 \cos\theta_1 \sin\theta_2 + mlr\dot{\theta}_1\dot{\theta}_2 \sin\theta_1 \cos\theta_2$$

$$+ ml^2\ddot{\theta}_2(\sin\theta_2)^2 + 2ml^2\dot{\theta}_2^2 \sin\theta_2 \cos\theta_2 + I\ddot{\theta}_2$$

$$+ mlr\dot{\theta}_1\dot{\theta}_2 \cos\theta_1 \sin\theta_2 - mlr\dot{\theta}_1\dot{\theta}_2 \sin\theta_1 \cos\theta_2$$

$$+ mgl \sin\theta_2 = M_2 \qquad \text{(eq. 3.23)}$$

Then, the second differential equation of the model can thus be expressed as follows:

$$mlr\ddot{\theta}_1 \cos(\theta_1 - \theta_2) + mlr\dot{\theta}_1^2 \sin(\theta_2 - \theta_1)$$

$$+ (I + ml^2)\ddot{\theta}_2 - mgl \sin\theta_2 = M_2 \qquad \text{(eq. 3.24)}$$

The model in [GUBI74] has an additional degree of freedom comparison to the model presented in this thesis which is variable leg length and does not have any ankle control torque. If the time derivative values of the leg length variable r are taken out, and the ankle torque M_1 is added to the dynamic equations in [GUBI74], it is seen that these equations are identical with the equations Eq. 3.17 and Eq. 3.23.

D. NEWTON-EULER FORMULATION

As discussed in the previous chapter, if the Lagrangian method is used, an increase in the number of degrees of freedom complicates the calculations exponentially for an articulated mechanism. Even though the number of variables are not many in this case, by considering that future research is likely to investigate more complex models, the Newton-Euler method looks as if a better approach for this kind of problems.

The model, presented in this chapter, can be divided into two parts to simplify the problem. These parts are the massless leg and the body itself.

1. The Newton-Euler Equations of the Massless Leg

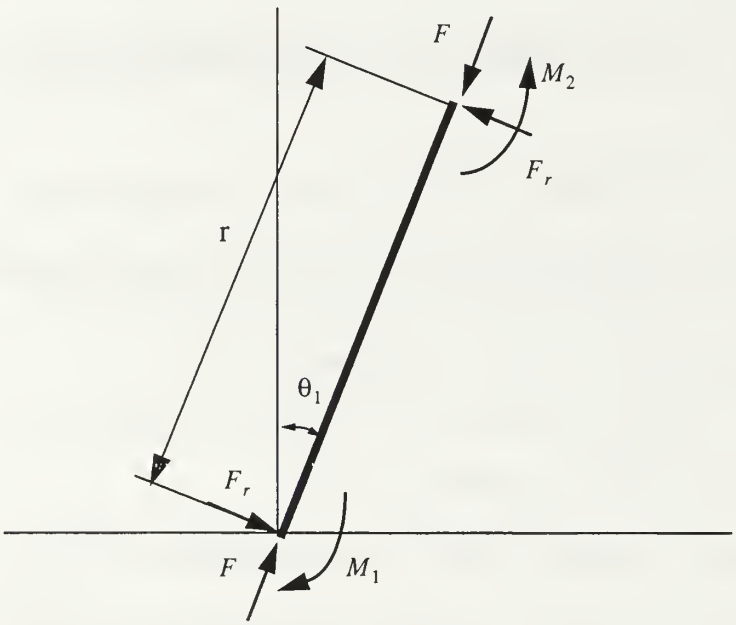

Figure 10: Free Body Diagram for The Massless Leg

Because the leg has no mass, it is only possible to talk about the static equilibrium of this body part. The equilibrium equations can be expressed as

$$\sum_{i=1}^{n} \vec{F}_i = 0 \qquad \text{(eq.3.25)}$$

and

$$\sum_{i=1}^{n} \vec{M}_i - F_r r = 0 \qquad \text{(eq. 3.26)}$$

where M_i, F_i and r are defined in Figure 10.

Equation Eq. 3.25 can also be written as

$$M_1 - M_2 - rF_r = 0 \qquad \text{(eq. 3.27)}$$

so the reaction force, F_r, is given by

$$F_r = \frac{M_1 - M_2}{r} \qquad \text{(eq. 3.28)}$$

2. The Newton-Euler Equations of the Body

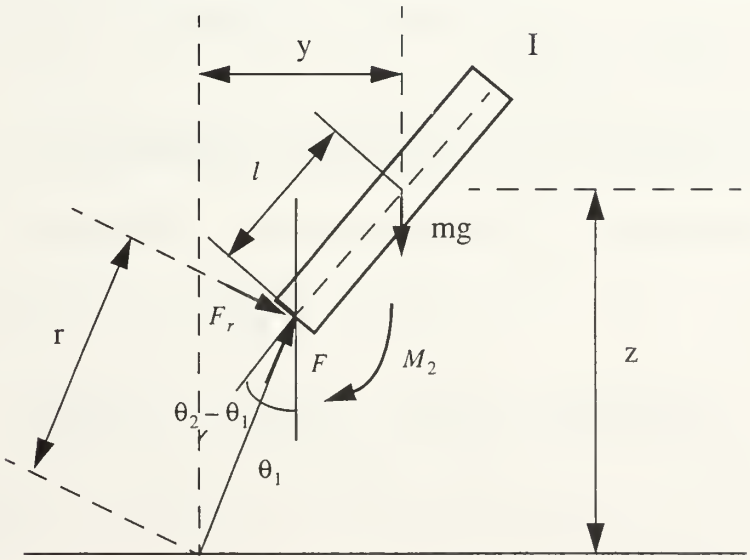

Figure 11: Free Body Diagram of The Body

The dynamic equations of the free body in the z and y directions can be derived as

$$m\ddot{z} = -mg + F\cos\theta_1 - F_r\sin\theta_1 \qquad \text{(eq. 3.29)}$$

and

$$m\ddot{y} = F\sin\theta_1 - F_r\cos\theta_1 \qquad \text{(eq. 3.30)}$$

For the angular motion of the free body, the dynamic equation of the body can be expressed as

$$I\ddot{\theta}_2 = M_2 - lF_r\cos(\theta_2 - \theta_1) + lF\sin(\theta_2 - \theta_1) \qquad \text{(eq. 3.31)}$$

3. Combining Body and Leg Equations

As shown in Figure 9, the geometric constraint equations for the model can be derived as

$$y = r\sin\theta_1 + l\sin\theta_2 \qquad \text{(eq. 3.32)}$$

and

$$z = r\cos\theta_1 + l\cos\theta_2 . \qquad \text{(eq. 3.33)}$$

The second time derivatives of Eq. 3.30 and Eq. 3.31 are as follows:

$$\ddot{y} = r\ddot{\theta}_1\cos\theta_1 - r\dot{\theta}_1^2\sin\theta_1 + l\ddot{\theta}_2\cos\theta_2 - l\dot{\theta}_2^2\sin\theta_2 \qquad \text{(eq. 3.34)}$$

and

$$\ddot{z} = -r\ddot{\theta}_1\sin\theta_1 - r\dot{\theta}_1^2\cos\theta_1 - l\ddot{\theta}_2\sin\theta_2 - l\dot{\theta}_2^2\cos\theta_2 . \qquad \text{(eq. 3.35)}$$

The constraint force F_r in equation Eq. 3.29 can be eliminated by using equation Eq. 3.26 resulting in the expression

$$I\ddot{\theta}_2 = M_2 - \frac{l(M_1 - M_2)\cos(\theta_2 - \theta_1)}{r} + lF\sin(\theta_2 - \theta_1) \qquad \text{(eq. 3.36)}$$

By combining Eq. 3.28 and 3.32, the following equation can be derived:

$$m(r\ddot{\theta}_1\cos\theta_1 - r\dot{\theta}_1^2\sin\theta_1 + l\ddot{\theta}_2\cos\theta_2 - l\dot{\theta}_2\sin\theta_2) =$$

$$= F\sin\theta_1 + F_r\cos\theta_1 \qquad \text{(eq. 3.37)}$$

On the other hand, after substituting $\ddot{z}$ with Eq. 3.33, Eq. 3.27 can be written as

$$m(-r\ddot{\theta}_1\sin\theta_1 - r\dot{\theta}_1^2\cos\theta_1 - l\ddot{\theta}_2\sin\theta_2 - l\dot{\theta}_2^2\cos\theta_2) =$$

$$= -mg + F\cos\theta_1 - F_r\sin\theta_1 \qquad \text{(eq. 3.38)}$$

The equation Eq. 3.34 can redefined as

$$\ddot{\theta}_2 I - Fl\sin(\theta_2 - \theta_1) = \frac{M_2(l+r) - lM_1}{r}\cos(\theta_2 - \theta_1) \qquad \text{(eq. 3.39)}$$

After reorganizing, Eq. 3.35 and Eq. 3.36 become

$$\ddot{\theta}_1 mr\cos\theta_1 + \ddot{\theta}_2 ml\cos\theta_2 - F\sin\theta_1 =$$

$$= mr\dot{\theta}_1^2\sin\theta_1 + ml\dot{\theta}_2^2\sin\theta_2 + \frac{M_1 - M_2}{r}\cos\theta_1 \qquad \text{(eq. 3.40)}$$

and

$$\ddot{\theta}_1 mr\sin\theta_1 + \ddot{\theta}_2 ml\sin\theta_2 + F\cos\theta_1 =$$

$$= -mr\dot{\theta}_1^2\cos\theta_1 - ml\dot{\theta}_2^2\cos\theta_2 + mg + \frac{M_1 - M_2}{r}\sin\theta_1 \qquad \text{(eq. 3.41)}$$

Equations Eq. 3.37, Eq. 3.38, and Eq. 3.39 can be expressed in matrix form as:

$$\begin{bmatrix} 0 & I & -l\sin(\theta_2-\theta_1) \\ mr\cos\theta_1 & ml\cos\theta_2 & -\sin\theta_1 \\ mr\sin\theta_1 & ml\sin\theta_2 & \cos\theta_1 \end{bmatrix} \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \\ F \end{bmatrix} =$$

$$\begin{bmatrix} \dfrac{M_2(l+r)-lM_1}{r}\cos(\theta_2-\theta_1) \\ mr\dot{\theta}_1^2\sin\theta_1 + ml\dot{\theta}_2^2\sin\theta_2 + \dfrac{M_1-M_2}{r}\cos\theta_1 \\ -mr\dot{\theta}_1^2\cos\theta_1 - ml\dot{\theta}_2^2\cos\theta_2 + mg + \dfrac{M_1-M_2}{r}\sin\theta_1 \end{bmatrix} \quad \text{(eq. 3.42)}$$

An alternative to this analytic formulation of the Newton-Euler formulation is to use the recursive approach explained in Chapter II of this thesis. Thus results in more complex logic, but in less computation, especially as the number of links in the dynamic model increases.

E. LINEARIZED ANALYSIS FOR CHOOSING GAINS

As mentioned in the previous sections, this system has two inputs, hip and ankle torques. The main purpose of postural control is to determine these input torques in order to maintain the upright orientation of the body. The approach taken in this thesis is linear state feedback control. The control equations, Eq. 3.5 and Eq. 3.6, are presented in the second section of this chapter. For the local control model, the problem is to compute the required gain values: k_{θ_1}, $k_{\dot{\theta}_1}$, k_{θ_2}, $k_{\dot{\theta}_2}$.

It is reasonable to drop quadratic velocity components for small motion linearized analysis [MCGH86]. Under this assumption, $\dot{\theta}_1^2$ and $\dot{\theta}_2^2$, components are removed from

all equations. The same assumption means that θ_1 and θ_2 can have only small values which allows the substitutions

$$\sin\theta_1 = \theta_1, \quad \text{(eq. 3.43)}$$

$$\sin\theta_2 = \theta_2, \quad \text{(eq. 3.44)}$$

$$\cos\theta_1 = 1, \quad \text{(eq. 3.45)}$$

and

$$\cos\theta_2 = 1. \quad \text{(eq. 3.46)}$$

With these considerations, equation Eq. 3.18 can be rewritten as

$$mr^2\ddot{\theta}_1 + lmr\ddot{\theta}_2 = mgr\theta_1 + M_1 - M_2. \quad \text{(eq. 3.47)}$$

The linearized version of equation Eq. 3.24 is

$$mlr\ddot{\theta}_1 + (I + ml^2)\ddot{\theta}_2 = mgl\theta_2 + M_2 \quad \text{(eq. 3.48)}$$

$\ddot{\theta}_2$ can be evaluated from equation Eq. 3.47 as

$$\ddot{\theta}_2 = \frac{mgr\theta_1 + M_1 - M_2 - mr^2\ddot{\theta}_1}{lmr} \quad \text{(eq. 3.49)}$$

If equation Eq. 3.49 is substituted in equation Eq. 3.48, the following equation is obtained

$$(mlr)^2\ddot{\theta}_1 + (I + ml^2)(mgr\theta_1 + M_1 - M_2 - mr^2\ddot{\theta}_1)$$

$$= rm^2gl^2\theta_2 + mlrM_2 \quad \text{(eq. 3.50)}$$

After reorganizing, equation Eq. 3.50, $\ddot{\theta}_1$ can be defined as

$$\ddot{\theta}_1 =$$

$$= \frac{l^2 m^2 rg\theta_2 - mgr(I + ml^2)\theta_1 - (I + ml^2)M_1 + (lmr + I + ml^2)M_2}{-Imr^2}$$

(eq. 3.51)

The angular acceleration, $\ddot{\theta}_1$, can be defined by using equation Eq. 3.47 as

$$\ddot{\theta}_1 = \frac{mgr\theta_1 + M_1 - M_2 - lmr\ddot{\theta}_2}{mr^2} \quad \text{(eq. 3.52)}$$

After substitution of $\ddot{\theta}_1$, equation Eq. 3.48 can be rewritten as

$$\frac{l}{r}(mgr\theta_1 + M_1 - M_2 - lmr\ddot{\theta}_2) + (I + ml^2)\ddot{\theta}_2 = mgl\theta_2 + M_2 \quad \text{(eq. 3.53)}$$

After reorganizing equation Eq. 3.53, $\ddot{\theta}_2$ can be define as

$$\ddot{\theta}_2 =$$

$$\frac{mglr\theta_2 - lmgr\theta_1 - lM_1 + (r + l)M_2}{rI} \quad \text{(eq. 3.54)}$$

The general linear state feedback equation is

$$\dot{x} = Ax + Bu \quad \text{(eq. 3.55)}$$

The state vector, x, is defined by equation Eq. 3.1. If the desired values for the angles are chosen as zero, the input vector, in equation Eq. 3.5, can be rewritten as

$$u = Kx \quad \text{(eq. 3.56)}$$

After substituting u according to the equation Eq. 3.56, Eq 3.53 can be written as

$$\dot{x} = (A + BK)x \qquad \text{(eq. 3.57)}$$

where

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \dfrac{mgr(I+ml^2)}{Imr^2} & 0 & \dfrac{l^2m^2rg}{Imr^2} & 0 \\ 0 & 0 & 0 & 1 \\ \dfrac{-lmgr}{Ir} & 0 & \dfrac{mglr}{Ir} & 0 \end{bmatrix}, \qquad \text{(eq. 3.58)}$$

and

$$B = \begin{bmatrix} 0 & 0 \\ \dfrac{I+ml^2}{Imr^2} & \dfrac{-(lmr+I+ml^2)}{Imr^2} \\ 0 & 0 \\ -\dfrac{l}{Ir} & \dfrac{r+l}{Ir} \end{bmatrix} \qquad \text{(eq. 3.59)}$$

For local feedback, K can be defined as follows, according to the equations Eq. 3.2, Eq. 3.6 and Eq 3.7

$$K = \begin{bmatrix} -K_{\theta_1} & -K_{\dot{\theta}_1} & 0 & 0 \\ 0 & 0 & -K_{\theta_2} & -K_{\dot{\theta}_2} \end{bmatrix} \qquad \text{(eq. 3.60)}$$

where desired angle values α_1 and α_2 are taken as zero. Then, by using equation Eq. 3.58, Eq. 3.59 and Eq. 3.60, $A + BK$ can be defined as

$$A + BK = \begin{bmatrix} 0 & 1 & 0 & 0 \\ M & N & O & P \\ 0 & 0 & 0 & 1 \\ Q & V & Y & Z \end{bmatrix} \quad \text{(eq. 3.61)}$$

where

$$M = \frac{mgr(I + ml^2) - K_{\theta_1}(I + ml^2)}{Imr^2} \quad \text{(eq. 3.62)}$$

$$N = \frac{-K_{\dot{\theta}_1}(I + ml^2)}{Imr^2} \quad \text{(eq. 3.63)}$$

$$O = \frac{-l^2 m^2 rg + K_{\theta_2}(lmr + I + ml^2)}{Imr^2} \quad \text{(eq. 3.64)}$$

$$P = \frac{K_{\dot{\theta}_2}(lmr + I + ml^2)}{Imr^2} \quad \text{(eq. 3.65)}$$

$$Q = \frac{-lmgr + lK_{\theta_1}}{rI} \quad \text{(eq. 3.66)}$$

$$V = \frac{K_{\dot{\theta}_1} l}{rI} \quad \text{(eq. 3.67)}$$

$$Y = \frac{mglr - K_{\theta_2}(r + l)}{rI} \quad \text{(eq. 3.68)}$$

$$Z = \frac{-K_{\dot{\theta}_2}(r + l)}{rI} \quad \text{(eq. 3.69)}$$

For a stable upright posture, all roots of the characteristic equation (eigenvalues) should have negative real parts [KUO95]. The characteristic equation of the closed loop system is [KUO95]

$$|sI-(A+BK)| = 0 \qquad \text{(eq. 3.70)}$$

After substituting the equation Eq. 3.60 the equation, Eq. 3.69 can be rewritten as

$$\begin{vmatrix} s & -1 & 0 & 0 \\ C & D & E & F \\ 0 & 0 & s & -1 \\ G & H & W & Y \end{vmatrix} = 0 \qquad \text{(eq. 3.71)}$$

where

$$C = \frac{-mgr(I+ml^2)+K_{\theta_1}(I+ml^2)}{Imr^2}, \qquad \text{(eq. 3.72)}$$

$$D = s+\frac{K_{\dot{\theta}_1}(I+ml^2)}{Imr^2}, \qquad \text{(eq. 3.73)}$$

$$E = \frac{l^2m^2rg-K_{\theta_2}(lmr+I+ml^2)}{Imr^2}, \qquad \text{(eq. 3.74)}$$

$$F = \frac{-K_{\dot{\theta}_2}(lmr+I+ml^2)}{Imr^2}, \qquad \text{(eq. 3.75)}$$

$$G = \frac{lmgr-K_{\theta_1}l}{rI}, \qquad \text{(eq. 3.76)}$$

$$H = \frac{-K_{\dot{\theta}_1} l}{rI}, \qquad \text{(eq. 3.77)}$$

$$W = \frac{-lmgr + K_{\theta_2}(r+l)}{rI}, \qquad \text{(eq. 3.78)}$$

$$Y = s + \frac{K_{\dot{\theta}_2}(r+l)}{rI} \qquad \text{(eq. 3.79)}$$

If the determinant of in the equation Eq. 3.69 is evaluated, the following polynomial is obtained:

$$\begin{aligned} & s^4(r^2 Im) \\ & + s^3\left(lrmK_{\dot{\theta}_2} + IK_{\dot{\theta}_1} + K_{\dot{\theta}_1} ml^2 + r^2 mK_{\dot{\theta}_2}\right) \\ & + s^2(IK_{\theta_1} - l^2 m^2 rg + ml^2 K_{\theta_1} - r^2 m^2 lg \\ & \qquad + r^2 mK_{\theta_2} + rmlK_{\theta_2} + K_{\dot{\theta}_1} K_{\dot{\theta}_2} - Imgr) \\ & + s(K_{\theta_1} K_{\dot{\theta}_2} - lmgK_{\dot{\theta}_1} + K_{\dot{\theta}_1} K_{\theta_2} - mgrK_{\dot{\theta}_2}) \\ & + \left(m^2 g^2 rl - mgrK_{\theta_2} - K_{\theta_1} lmg + K_{\theta_1} K_{\theta_2}\right) = 0 \end{aligned} \qquad \text{(eq. 3.80)}$$

It is clearly a very hard problem to determine the required gain values in equation Eq. 3.80, which result in all roots having negative real parts. Instead, a experimental solution was found by trying some gain values on the computer model. It was found that the following values produce an upright stable body posture:

$$K_{\theta_1} = 25000, \quad \text{(eq. 3.81)}$$

$$K_{\dot{\theta}_1} = 2500, \quad \text{(eq. 3.82)}$$

$$K_{\theta_2} = 4000, \quad \text{(eq. 3.83)}$$

$$K_{\dot{\theta}_2} = 400 \quad \text{(eq. 3.84)}$$

F. SUMMARY

In this chapter, a new human dynamic model, which is made of a body and a massless leg, is introduced. The dynamic differential equations of the model are derived by using two different methods: the Lagrangian and the Newton-Euler methods. The last section of the chapter discusses the small motion linearized analysis of the system. It is explained how the gain constants for the input torques should be chosen.

The next chapter will explain the implementation of computer models based on the knowledge presented in the second and third chapters of this thesis.

IV. COMPUTER MODELS

A. INTRODUCTION

In the second chapter of this thesis, it is pointed out that there exist two major methods to simulate human motion simulation: kinematic and dynamic models. The present chapter first presents stepping algorithms for a human figure implementation which is developed by using a kinematic model. The second chapter also discusses the general approaches for dynamic simulation of human motion. Based on this knowledge, the third chapter investigates the mathematical representation of a dynamic human model. The present chapter also introduces the computer implementations of these models.

B. KINEMATIC COMPUTER MODEL (DYNAMAN)

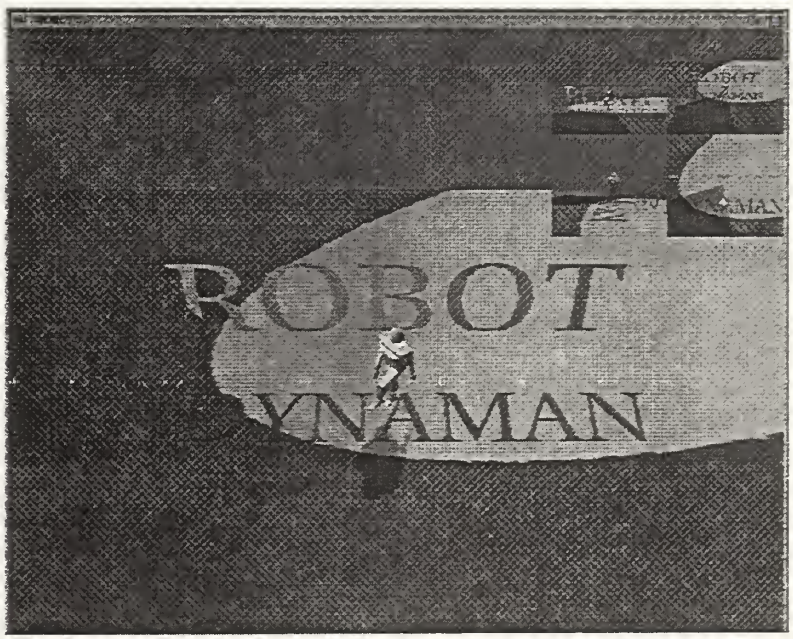

Figure 12: Kinematic Computer Model: Dynaman

The kinematic model presented in this section is made of fifteen separate body parts as shown in Figure 13.

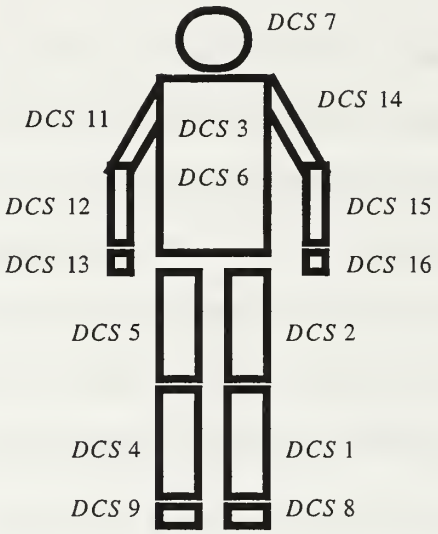

Figure 13: The Body Parts and Corresponding Dynamic Coordinate Systems (DCS) of Dynaman

The purpose of the simulation is to create a computer representation of a stepping human. Under this requirement, it is clear that the main concern is leg and foot motion. The other body parts are synchronized according to the legs.

Inverse kinematics is chosen for computing leg motion. Inverse kinematic equations take the position of the end effector (foot) as the input and computes each joint angle of the leg. The points which describe the position of the foot in space for a full gait period produce a path. The question is to define this path as a mathematical function of time. This function, of course, should be developed with the knowledge of the geometry of the environment, such as height of each stair.

Forward kinematics could have been chosen for the same problem. In that case, joint angles of the leg would be the input to the forward kinematic equations, and the position of the end effector would be calculated. For this second case, there should exist a

feedback system in order to input the constraints of the environment to the model. A collision detection mechanism between the foot and the floor seems to be appropriate to this approach [GOET94].

1. Inverse Kinematic Equations for Three Link Planar Manipulator

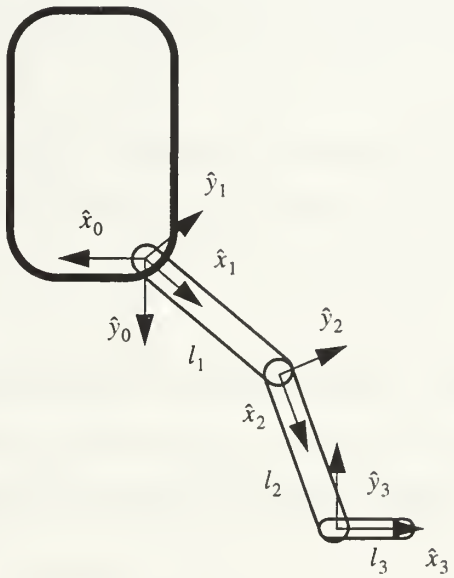

Figure 14: Three Link Planar Manipulator

The leg of Dynaman can be thought as a three link planar manipulator which is made of the upper leg, the lower leg, and the foot. The frames which are attached to the links are shown in Figure 14. The general form of the transformation matrix to represent a point in the frame $i-1$ which is defined in frame i is [CRAI89]

$$^{i-1}_{i}T =$$

$$= \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i\cos\alpha_{i-1} & \cos\theta_i\cos\alpha_{i-1} & -\sin\alpha_{i-1} & -d_i\sin\alpha_{i-1} \\ \sin\theta_i\sin\alpha_{i-1} & \cos\theta_i\sin\alpha_{i-1} & \cos\alpha_{i-1} & d_i\cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(eq. 4.1)}$$

where a, α, θ, and d are the link parameters as defined in the second chapter of this thesis. The link parameters for the presented model are defined in Table 1.

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	0	l_1	0	θ_2
3	0	l_2	0	θ_3

Table 1. Link Parameters of the Leg [CRAI89]

In this model, all z_i axes are parallel, and all x_i axes are in the same plane. That is why all the α_{i-1} and d_i values are zero. The parameters l_1 and l_2 are the link lengths of the first and second links. The parameter l_3 defines the position of the toe point. Since it is in the end effector frame, it is not included in the link parameters.

According to equation Eq. 4.1 and Table 1, the transformation matrices for the neighbor links are

$$^0_1T = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(eq. 4.2)}$$

$$
{}^{1}_{2}T = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & l_1 \\ \sin\theta_2 & \cos\theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(eq. 4.3)}
$$

$$
{}^{2}_{3}T = \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & l_2 \\ \sin\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(eq. 4.4)}
$$

The transformation matrix between the first and the last links can be derived by using the product

$$
{}^{0}_{3}T = {}^{0}_{1}T \; {}^{1}_{2}T \; {}^{2}_{3}T \qquad \text{(eq. 4.5)}
$$

If the matrix multiplications are executed after substituting equations Eq. 4.2, Eq. 4.3, and Eq. 4.4 into the equation Eq. 4.5, the following transformation matrix will be obtained:

$$
{}^{0}_{3}T =
$$

$$
\begin{bmatrix} \cos(\theta_1+\theta_2+\theta_3) & -\sin(\theta_1+\theta_2+\theta_3) & 0 & l_1\cos\theta_1 + l_2\cos(\theta_1+\theta_2) \\ \sin(\theta_1+\theta_2+\theta_3) & \cos(\theta_1+\theta_2+\theta_3) & 0 & l_1\sin\theta_1 + l_2\sin(\theta_1+\theta_2) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

(eq. 4.6)

If the method in [CRAI89] is used, the transformation matrix of the end effector according to the base link can be defined as

$$ {}^{0}_{3}T = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & x \\ \sin\theta & \cos\theta & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(eq. 4.7)} $$

where the position components, x and y, and the orientation, θ, of the end effector according to the base link are known. The following equations can be derived from equations Eq. 4.6 and Eq. 4.7:

$$ \cos\theta = \cos(\theta_1 + \theta_2 + \theta_3) \qquad \text{(eq. 4.8)} $$

$$ \sin\theta = \sin(\theta_1 + \theta_2 + \theta_3) \qquad \text{(eq. 4.9)} $$

$$ x = l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) \qquad \text{(eq. 4.10)} $$

$$ y = l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2) \qquad \text{(eq. 4.11)} $$

If the squares of equations Eq. 4.10 and Eq 4.11 are added, the following equation is obtained:

$$ x^2 + y^2 = l_1^2 + l_2^2 + 2 l_1 l_2 \cos\theta_2 \qquad \text{(eq. 4.12)} $$

In equation Eq. 4.12, the only unknown is θ_2 and it can be defined as

$$ \theta_2 = \operatorname{acos}\left(\frac{x^2 + y^2 - {l_1}^2 - {l_2}^2}{2 l_1 l_2} \right) \qquad \text{(eq. 4.13)} $$

After having found θ_2, equations Eq. 4.10 and 4.11 can be can be written as

$$ x = k_1 \cos\theta_1 - k_2 \sin\theta_1 \qquad \text{(eq. 4.14)} $$

and

$$y = k_1 \sin\theta_1 + k_2 \cos\theta_1 \qquad \text{(eq. 4.15)}$$

where

$$k_1 = l_1 + l_2 \cos\theta_2 \qquad \text{(eq. 4.16)}$$

and

$$k_2 = l_2 \sin\theta_2 \qquad \text{(eq. 4.17)}$$

After following the steps in [CRAI89], θ_1 is defined by the equation

$$\theta_1 = \text{atan}\left(\frac{y}{x}\right) - \text{atan}\left(\frac{k_2}{k_1}\right) \qquad \text{(eq. 4.18)}$$

Then foot angle can then be calculated as

$$\theta_3 = \theta - \theta_1 - \theta_2 \qquad \text{(eq. 4.19)}$$

2. Link Descriptions in the Computer Model

The computer model has been developed in the IRIS Performer™ application environment. This environment allows the programer to create a *Dynamic Coordinate Systems (DCS)*. DCS lets the programer to change the transformation of the objects which are attached to that DCS node. The method followed in this application is to attach each body part to a DCS node and connect all the DCS nodes in a tree structure which creates a hierarchical structure. For example, if the upper leg is rotated for some degrees, all the body parts under that DCS node, lower leg and foot, follow this rotation. DCS tree structure of the whole body is shown in Figure 13 and Figure 15.

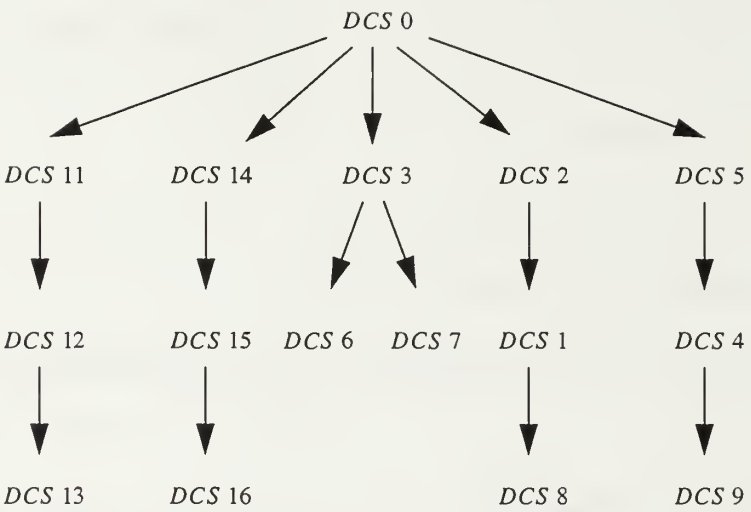

Figure 15: Dynamic Coordinate System (DCS) Hierarchy Tree of Dynaman

The body parts of the Dynaman are taken from an other model developed in OpenGL^{R} by Will Frey at the Naval Postgraduate School [FREY96]. However, while Frey used body segment Euler angles relative to an Earth fixed reference system, as described above, the work of this thesis is based on joint angles. The coordinates of the polygons, taken from the other model, which produces the body parts, are loaded to the application by using *poly* format files.

3. Stepping Algorithms

The inverse kinematics model takes position and orientation of the end effector and computes each joint angle. Then, the question is to define the path of the end effector (foot) which is the input to the inverse kinematics model. Two separate algorithms are developed to define the path: stepping forward and stepping upward algorithms.

a. Stepping Forward Algorithm

The leg, without bending at the knee, can be assumed as a simple swinging bar on a circular path which is shown as P_1 in Figure 16. The angle between the vertical

and the swinging bar is defined as α, which takes values between -20 and +20 degrees during the gait cycle. In reality, the knees are bent during stepping. Larger values of α introduce larger bending angles at the knee. When α is equal to zero, knee bending angle should also be zero. This requirement is introduced by

$$P_2 = P_1 \cos\alpha \qquad \text{(eq. 4.20)}$$

There is a difference between the supporting and the recovery leg motions. While the supporting leg always touches the ground, the recovery leg should be moved without touching the ground. This requirement is added to the model by using the following equation for the recovery leg

$$P_3 = 0.95P_2 \qquad \text{(eq. 4.21)}$$

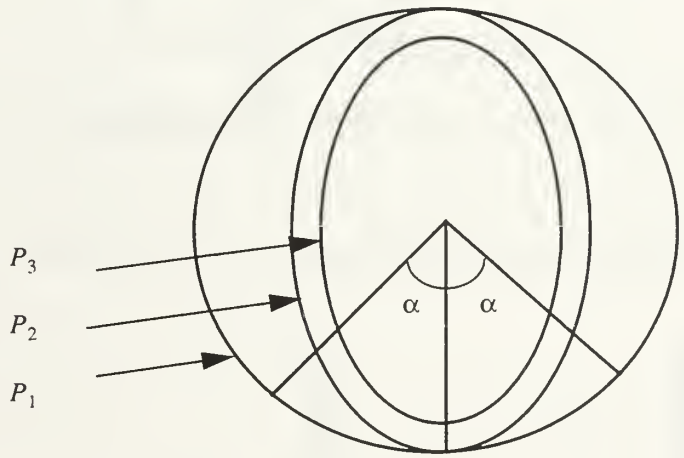

Figure 16: Forward Stepping Algorithm

No kinematic computation is needed for the arms. They are synchronized with the legs. The arm rotation is not exactly the same as leg rotation; a scale factor is applied for a more realistic looking arm motion. Another property of the model is the rotation of the upper body around the vector described by the general direction of motion. This motion is caused by the roll moments which are generated by the nature of biped

locomotion since the supporting legs have offset from the center of mass on opposite sides and are alternating during each step cycle.

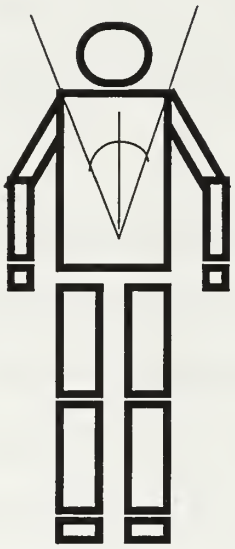

Figure 17: Upper Body Rotation

The distance from the upper body to the ground is not constant. When the α parameter has larger values, the whole body gets closer to the ground. This property is implemented by moving the whole body vertically with a cosine function of time.

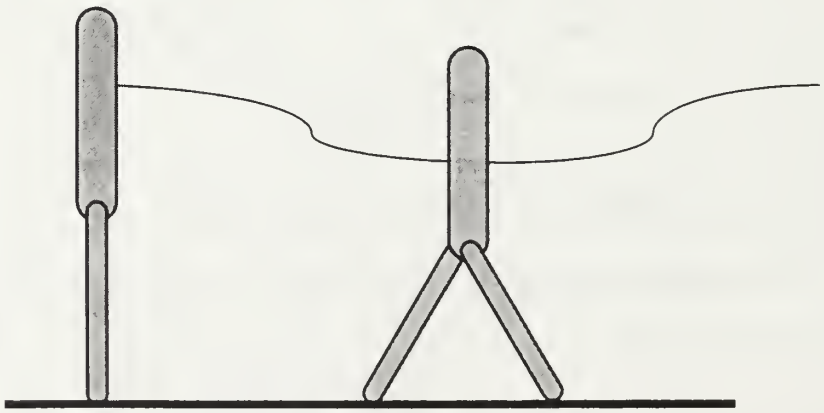

Figure 18: Change in the Height of the Body During One Gait Cycle

The height of the upper body is calculated by

$$Height = 0.08\cos(2\pi ft) \qquad \text{(eq. 4.22)}$$

b. Stepping Upward Algorithm

The stepping up algorithm has a major difference from the forward stepping which is caused by the need for increase in body elevation

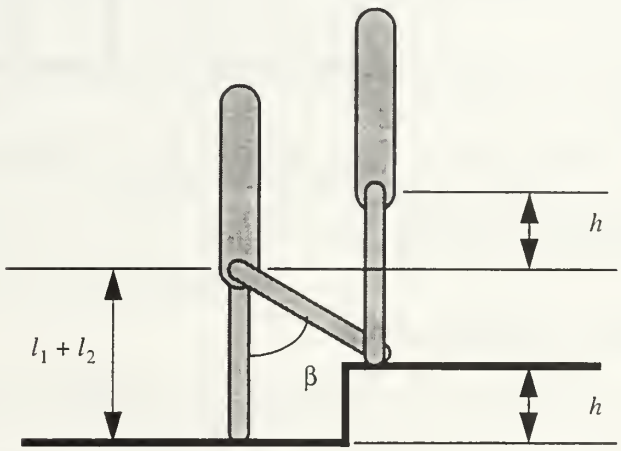

Figure 19: The Amount of Elevation For Stepping Up

The needed elevation for stepping up is

$$h = (l_1 + l_2) - (l_1 + l_2)\cos\beta. \qquad \text{(eq. 4.23)}$$

This elevation can only be handled by raising the foot to a higher position in the front. This causes the change which is shown in Figure 20. The values of α change from -10 to +30 degrees for the gait cycle.

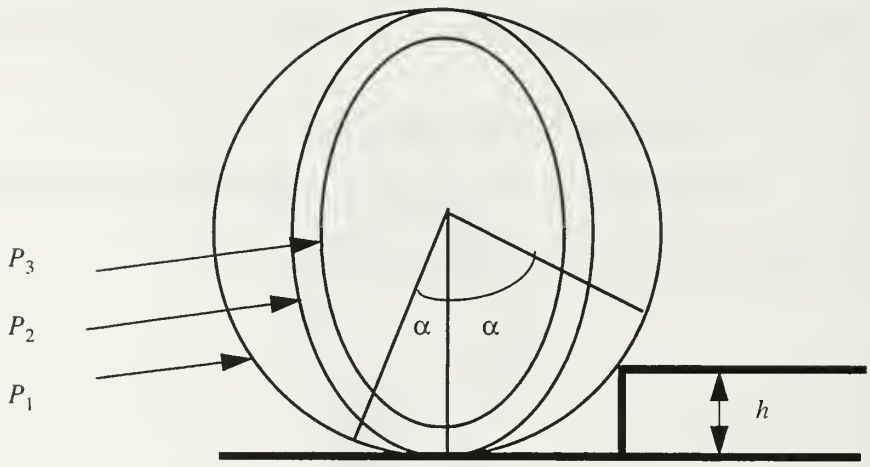

Figure 20: Stepping Up Algorithm

C. DYNAMIC COMPUTER MODELS

1. Newton-Euler Rigid Body Class

The Dynamic models in this thesis are developed in ANSI Common Lisp. The main class for the dynamic simulation is *rigid-body* class which was written by Professor McGhee. This Lisp code for this class is included as Appendix A of this thesis. This class defines a free rigid body in space. The major method of the rigid-body class is *update-rigid-body* which updates the *posture* vector which includes the position and the orientation of the rigid body in an earth coordinate system. This method converts body velocity rates to earth velocity rates to update the six element posture vector. Euler integration is used to update body velocities by using body velocity growth rates. The body velocity growth rates result from applied forces and torques on the body. The linear velocity growth rate is computed by [FRAN69]

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} rv - qw + \dfrac{f_x}{m} - g\sin\theta \\ pw - ru + \dfrac{f_y}{m} + \cos\theta\sin\phi \\ qu - pv + \dfrac{f_z}{m} + g\cos\theta\cos\phi \end{bmatrix} \qquad \text{(eq. 4.24)}$$

where f_x, f_y, f_z is the force vector applied to the body, (u, v, w) is the linear velocity vector, (p, q, r) is the rotational velocity vector, (ϕ, θ, ψ) is the body orientation defined in Euler angles, m is the mass, and g is the gravitational acceleration. All these vectors are defined in a body *principal axis* coordinate system [FRAN69]. The angular velocity growth rate is computed by

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \dfrac{[(I_{yy} - I_{zz})qr + L]}{I_{xx}} \\ \dfrac{[(I_{zz} - I_{xx})rp + M]}{I_{yy}} \\ \dfrac{[(I_{xx} - I_{yy})pq + N]}{I_{zz}} \end{bmatrix} \qquad \text{(eq. 4.25)}$$

where (L, M, N) is the torque vector applied to the body, I_{xx}, I_{yy}, and I_{zz} are mass moments of inertia, (p, q, r) is the rotational velocity vector [FRAN69]. In summary, the *update-rigid-body* method calculates the new position and orientation of the free rigid body in an earth coordinate system according to the applied forces and torques to the body. An other important method is *move* which takes the displacement components and orientation Euler angles as input and moves the body according to the inputs.

2. Numerical Integration Methods

Two different integration methods are used in the dynamic modeling implementations: *Euler* and *Huen* integration. The *Euler* integration approach can be formalized as [KREY88]

$$x_{n+1} = x_n + f(x_n, t_n)\Delta t \qquad \text{(eq. 4.26)}$$

where

$$\dot{y} = f(x_n, t) \qquad \text{(eq. 4.27)}$$

and Δt is a constant increment in the independent variable, t.

The *Heun* integration formula can be defined with the equation [KREY88]

$$x_{n+1} = x_n + \frac{1}{2}[f(x_n, t_n) + f(x_{n+1}{}^*, t_{n+1})]\Delta \qquad \text{(eq. 4.28)}$$

where

$$x_{n+1}{}^* = x_n + f(x_n, t_n)\Delta t. \qquad \text{(eq. 4.29)}$$

The symbols Δt and $f(x_n, t_n)$ have the same definitions for Heun integration as in Euler integration.

3. Dynamic Inverted Pendulum Simulations

Three different dynamic inverted pendulum models are implemented in this thesis:

a. A Single Link Single Rigid Body with Newton-Euler

The second implementation simulates an inverted pendulum by using Newton-Euler formulation of the dynamic equations of the system. The constraint forces from the dynamic equations and the control torque are the inputs of the rigid-body class. The 3x3 matrix multiplication form of the system equations is as follows:

$$\begin{bmatrix} I & -l\cos\theta & l\sin\theta \\ ml\cos\theta & 1 & 0 \\ ml\sin\theta & 0 & -1 \end{bmatrix} \begin{bmatrix} \ddot{\theta} \\ {}^{E}F_{x} \\ {}^{E}F_{z} \end{bmatrix} = \begin{bmatrix} M \\ \dot{\theta}^{2} ml\sin\theta \\ mg - \dot{\theta}^{2} ml\cos\theta \end{bmatrix} \qquad \text{(eq. 4.30)}$$

where all the variables are defined as in Figure 21.

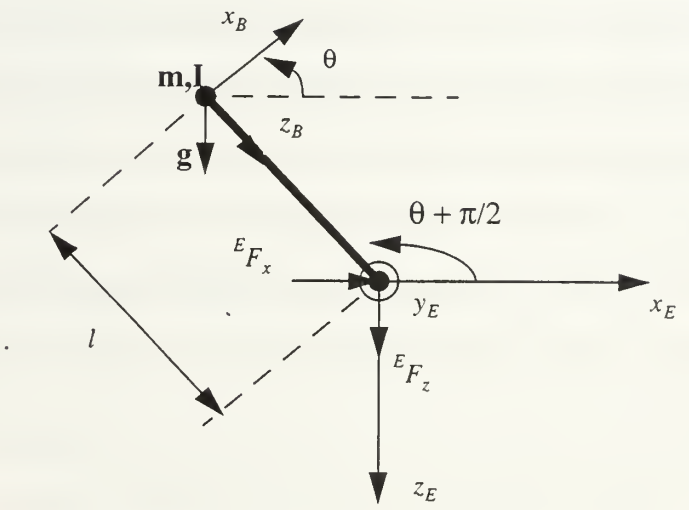

Figure 21: Inverted Pendulum with Constraint Forces

The system has only one degree of freedom. A suitable state vector is

$$x = [\theta, \dot{\theta}] \qquad \text{(eq. 4.31)}$$

To assume $M = 0$, makes the system behave like a natural inverted pendulum without control. By using linear state feedback, the control moment can be determined as

$$M = -K_{\phi}\phi - K_{\dot{\phi}}\dot{\phi} \qquad \text{(eq. 4.32)}$$

where K_{ϕ} and $K_{\dot{\phi}}$ are control gain variables. Suitable values for K_{ϕ} and $K_{\dot{\phi}}$ can be found analytically for small motion linearization of this system because its characteristic equation is quadratic. The Lagrangian formulation, Eq 2. 27,is most useful for this purpose.

b. Massless Leg and a Single Rigid Body with Lagrangian

The second implementation simulates a two link structure which is presented in the third chapter of this thesis. The dynamic equations of the system are defined with Eq. 3.18 and Eq. 3.24 which are derived by using Lagrangian method. The linear state feedback control equations are given by the equations Eq. 3.6 and Eq. 3.7. The state vector is defined as Eq. 3.1.

c. Massless Leg and a Single Rigid Body with Newton-Euler

The third implementation simulates the same two link structure which is simulated in the second simulation. However the Newton-Euler method is used to derive the dynamic equation instead of the Lagrangian. The 3x3 matrix form of the dynamic equations are given by Eq. 3.42. The linear state feedback control equations and the state vector are defined as the same as in the Lagrangian version of the simulation.

D. SUMMARY

This chapter presents forward and upward kinematic stepping algorithms, of an human model which is developed at the IRIS Performer^{TM} application environment by using C++. The second part of the chapter explains three different implementations to simulate various inverted pendulum models developed in Lisp. The next chapter discuses the results of these simulations.

V. RESULTS OF COMPUTER SIMULATIONS

A. INTRODUCTION

In this chapter, the results of the dynamic simulations are presented. The second part of the chapter contains frames from the kinematic simulation of Dynaman which show the model stepping forward and upward.

B. DYNAMIC SIMULATIONS

1. A Single Link Single Rigid Body with Newton-Euler Method

Figure 22, Figure 23, and Figure 24 show the behavior of the inverted pendulum with a control torque at the pivot point. The dynamic equations of the system are derived by using the Newton-Euler Method. The mass of the rigid body, m, is 100 lb. The body rotary inertia, I, is 900 lb. ft.^{2}. The length of the inverted pendulum, l, is 3 ft. The gravitational acceleration, g, is 32.2185 ft. / sec.^{2}. The gain values for the control torque are

$$K_{\theta} = 10000 \qquad \text{(eq. 5.1)}$$

$$K_{\dot{\theta}} = 2000 \qquad \text{(eq. 5.2)}$$

The initial state vector is

$$x = (1, 0) \qquad \text{(eq. 5.3)}$$

and the state vector in steady state is

$$x = (0, 0) \qquad \text{(eq. 5.4)}$$

Figure 25 shows inverted pendulum orientation change in time. Euler integration with a time step of 0.02 seconds was used for the results shown in Figure 22-25.

Figure 22: Initial Position and Orientation of the Inverted Pendulum

Figure 23: Inverted Pendulum Moving to the Upright Orientation

Figure 24: Inverted Pendulum After Steady State

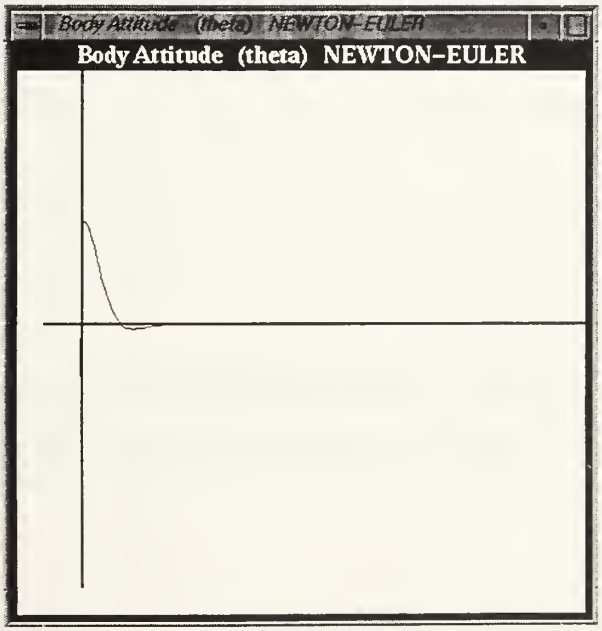

Figure 25: Body Attitude Response of the Inverted Pendulum

2. Massless Leg and a Single Rigid Body with Lagrangian Method

Figure 26, Figure 27, Figure 28, and Figure 29 show the behavior of the two link inverted pendulum with control torques at the hip and at the ankle. The z axis of the body is drawn to be able to observe the body attitude and the leg angle separately. The dynamic equations of the system is derived by using the Lagrangian Method. The mass of the rigid body, m, is 100 lb. The body rotary inertia, I, is 100 lb. ft.^{2}. The length of the leg, r, is 3 ft. The length of the rigid body, l, is 0.5 ft. The gravitational acceleration, g, is 32.2 ft. / sec.^{2}. The gain values for the control torques are

$$K_{\theta_1} = 25000 \qquad \text{(eq. 5.5)}$$

$$K_{\dot{\theta}_1} = 2500 \qquad \text{(eq. 5.6)}$$

$$K_{\theta_2} = 40000 \qquad \text{(eq. 5.7)}$$

$$K_{\dot{\theta}_2} = 400 \qquad \text{(eq. 5.8)}$$

The initial state vector is

$$x = (0.5, 0, 1, 0) \qquad \text{(eq. 5.9)}$$

and the state vector in steady state is

$$x = (0, 0, 0, 0) \qquad \text{(eq. 5.10)}$$

Figure 30 and Figure 31 show body attitude and leg angle changes in time. Heun integration with a time step of 0.01 seconds was used for the results shown in Figure 26-31.

Figure 26: Initial Orientation of the Two Link Inverted Pendulum (Lagrangian)

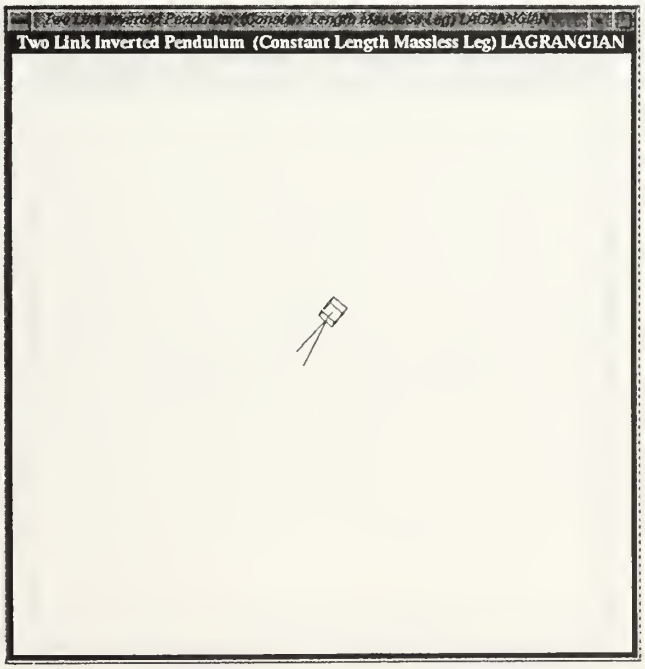

Figure 27: Two Link Inverted Pendulum Moving to the Upright Orientation by Control Torques(Lagrangian)

Figure 28: Two Link Pendulum Recovering the Negative Orientations (Lagrangian)

Figure 29: Two Link Inverted Pendulum After Steady State (Lagrangian)

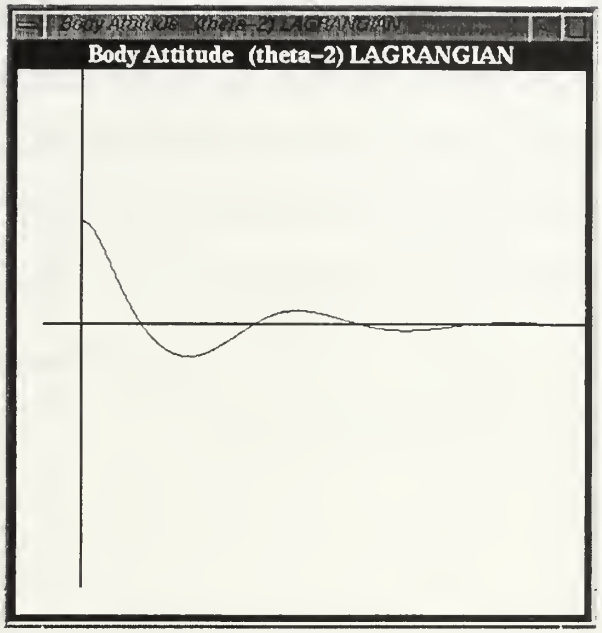

Figure 30: Body Attitude Response of the Two Link Inverted Pendulum (Lagrangian)

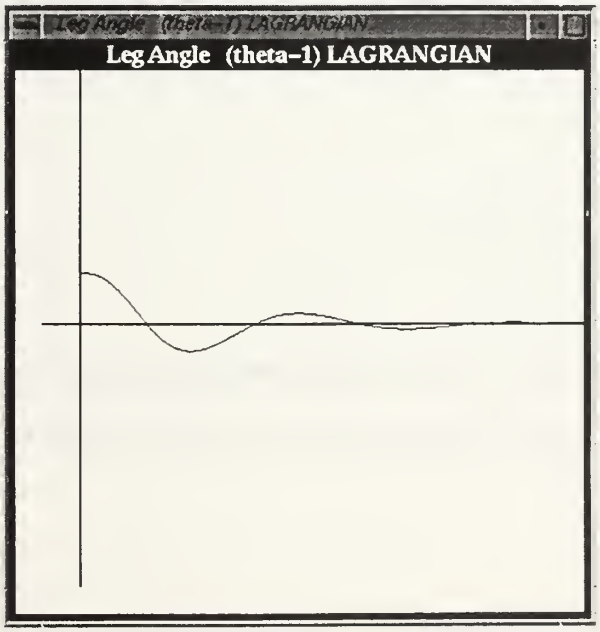

Figure 31: Leg Angle Response of the Two Link Inverted Pendulum (Lagrangian)

3. Massless Leg and a Single Rigid Body with Newton-Euler Method

Figure 32, Figure 33, Figure 34, and Figure 35 show the behavior of the two link inverted pendulum with control torques at the hip and at the ankle. Again, the z axis of the body is drawn to be able to observe the body attitude and the leg angle separately. The dynamic equations of the system are derived using the Newton-Euler Method. The mass of the rigid body, m, is 100 lb. The body rotary inertia, I, is 100 lb. ft.^{2}. The length of the leg, r, is 3 ft. The length of rigid body, l, is 0.5 ft. The gravitational acceleration, g, is 32.2 ft. / sec.^{2}. The gain values for the control torques are

$$K_{\theta_1} = 25000 \qquad \text{(eq. 5.11)}$$

$$K_{\dot{\theta}_1} = 2500 \qquad \text{(eq. 5.12)}$$

$$K_{\theta_2} = 40000 \qquad \text{(eq. 5.13)}$$

$$K_{\dot{\theta}_2} = 400 \qquad \text{(eq. 5.14)}$$

The initial state vector is

$$x = (0.5, 0, 1, 0) \qquad \text{(eq. 5.15)}$$

and the state vector in steady state is

$$x = (0, 0, 0, 0) \qquad \text{(eq. 5.16)}$$

Figures 32 through 37 show body attitude and leg angle changes in time. Heun integration with a time step of 0.01 seconds was used for these results.

Figure 32: Initial Orientation of the Two Link Inverted Pendulum (Newton-Euler)

Figure 33: Two Link Pendulum Moving to the Upright Orientation (Newton-Euler)

Figure 34: Two Link Pendulum Recovering the Negative Orientations (Newton-Euler)

Figure 35: Two Link Inverted Pendulum In Steady State (Newton-Euler)

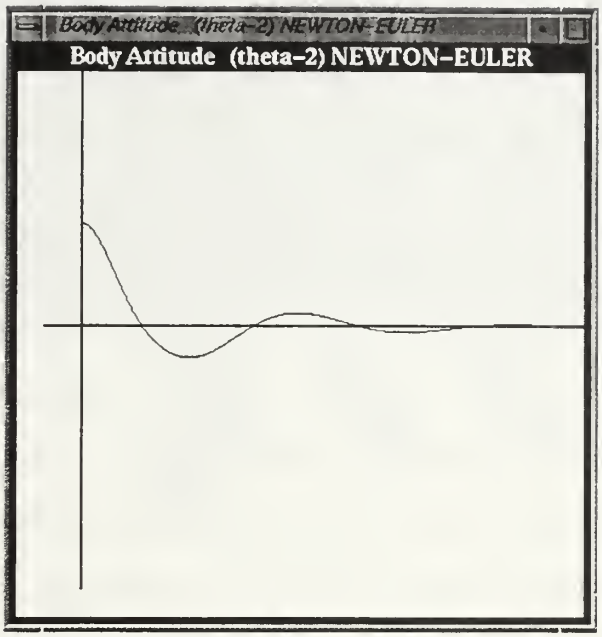

Figure 36: Body Attitude Time Response of the Two Link Inverted Pendulum (Newton-Euler)

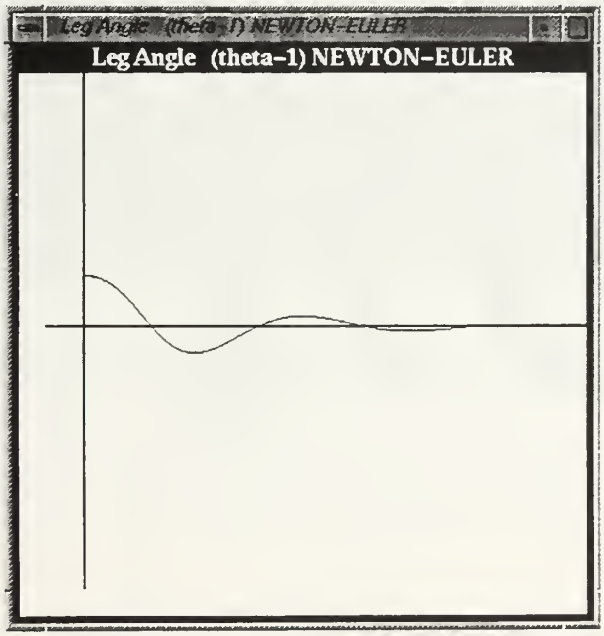

Figure 37: Leg Angle Response of the Two Link Inverted Pendulum (Newton-Euler)

C. RESULTS OF THE DYNAMIC SIMULATION

Euler integration was used for the single link inverted pendulum simulation. For the two link inverted pendulum simulations, Heun integration is chosen. Heun is a more accurate method than Euler, because it converges quadratically as the step size is decreased, while Euler integration converges only linearly.

As seen in the previous section, the Newton-Euler and Lagrangian solutions of the two link inverted pendulum problem give identical results. However, the running times of these simulations are not the same. The time needed to complete the Newton-Euler version of the simulation is 20 percent longer than the Lagrangian version. This is an expected consequence of the matrix inversion which takes place in the Newton-Euler version. The Lagrangian version of the problem computes the accelerations without matrix inversion. However, it is hard to derive dynamic equations using Lagrangian methods for more complex models with higher degrees of freedom. Moreover, due to the complexity of such equations, it is suspected that Newton-Euler models for postural control may run faster for more complex systems. This will certainly be true if O(n) methods are used [MCMI95].

D. KINEMATIC SIMULATION OF STEPPING DYNAMAN

1. Stepping Forward Algorithm

Figures 38 through 43 are six frames from the kinematic model simulation. These six frames show one step cycle of Dynaman during forward stepping.

Figure 38: Forward Stepping (1)

Figure 39: Forward Stepping (2)

Figure 40: Forward Stepping (3)

Figure 41: Forward Stepping (4)

Figure 42: Forward Stepping (5)

Figure 43: Forward Stepping (6)

2. Stepping Upward Algorithm

Figure 44 through 48 are five frames from the kinematic model simulation. These five frames show one step cycle of the Dynaman while stepping upward.

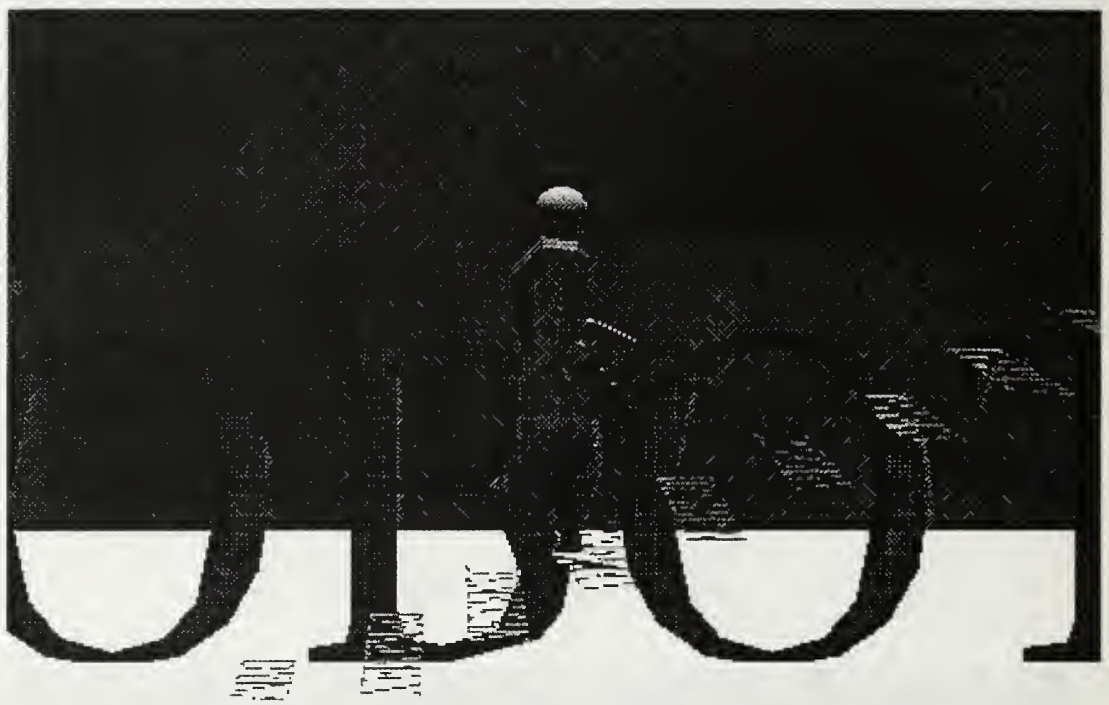

Figure 44: Upward Stepping (1)

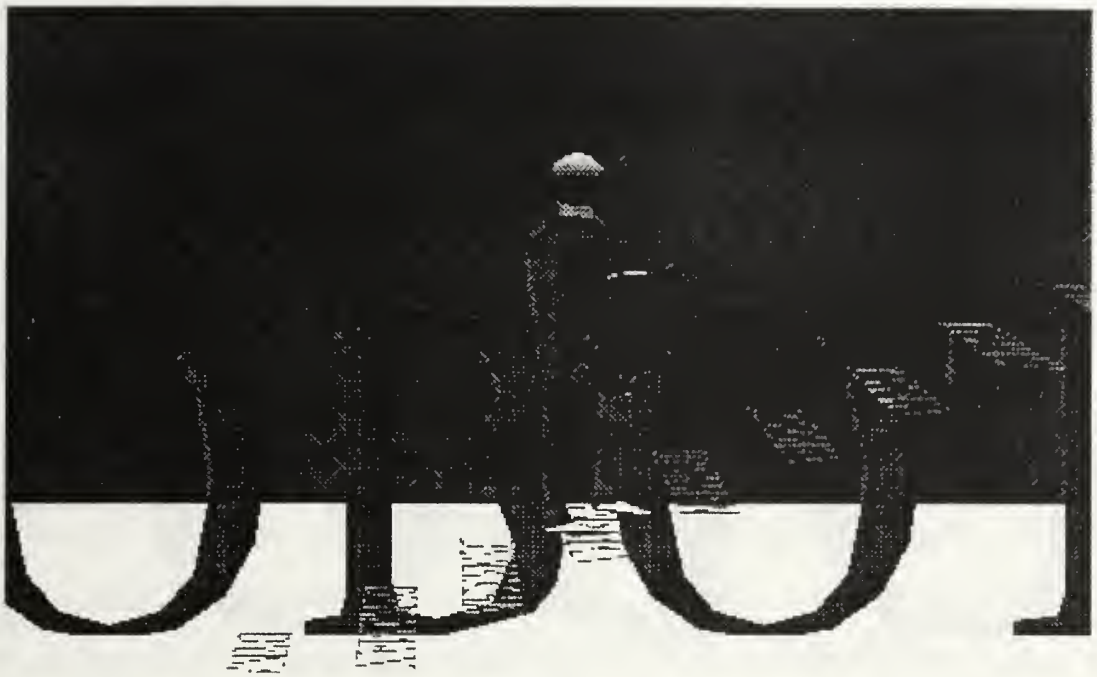

Figure 45: Upward Stepping (2)

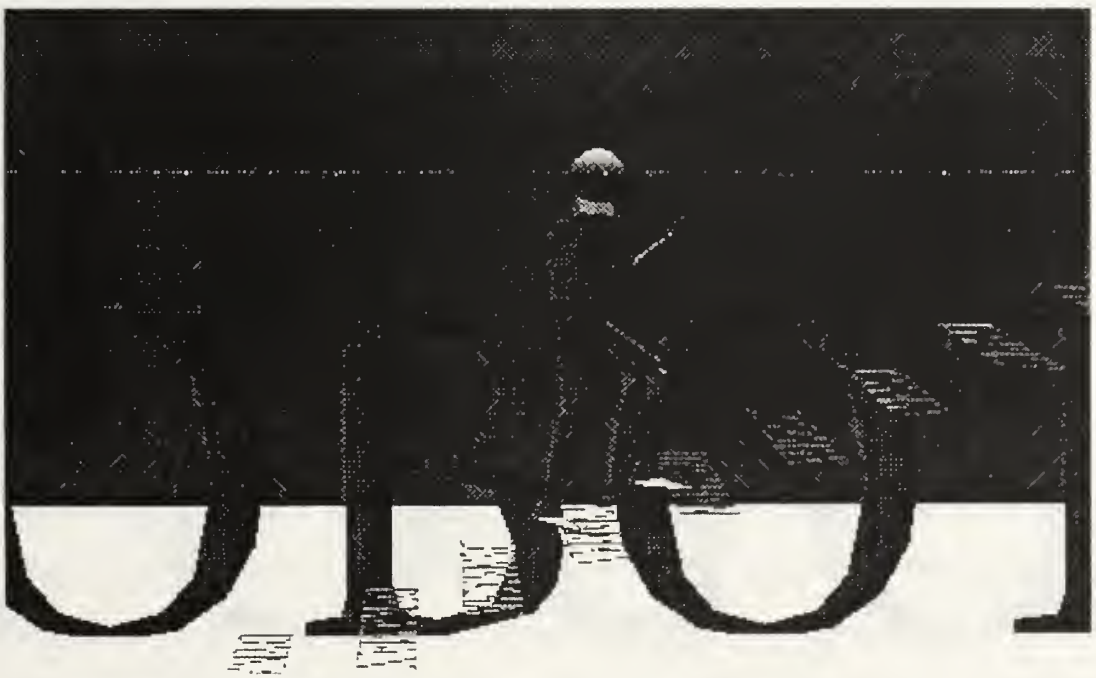

Figure 46: Upward Stepping (3)

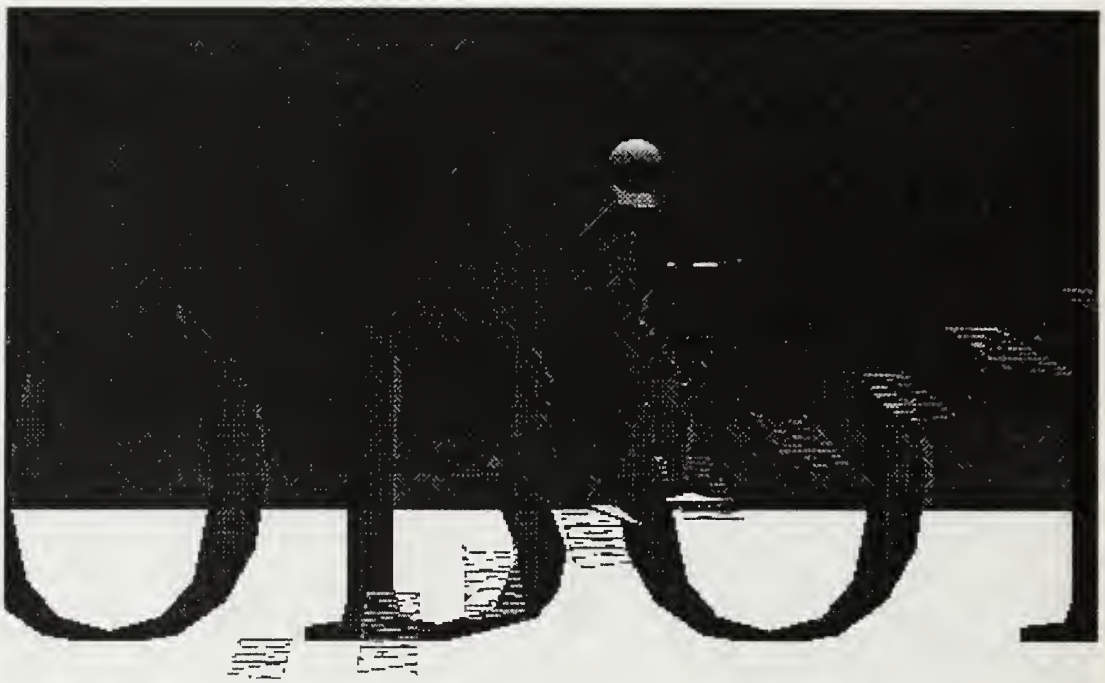

Figure 47: Upward Stepping (4)

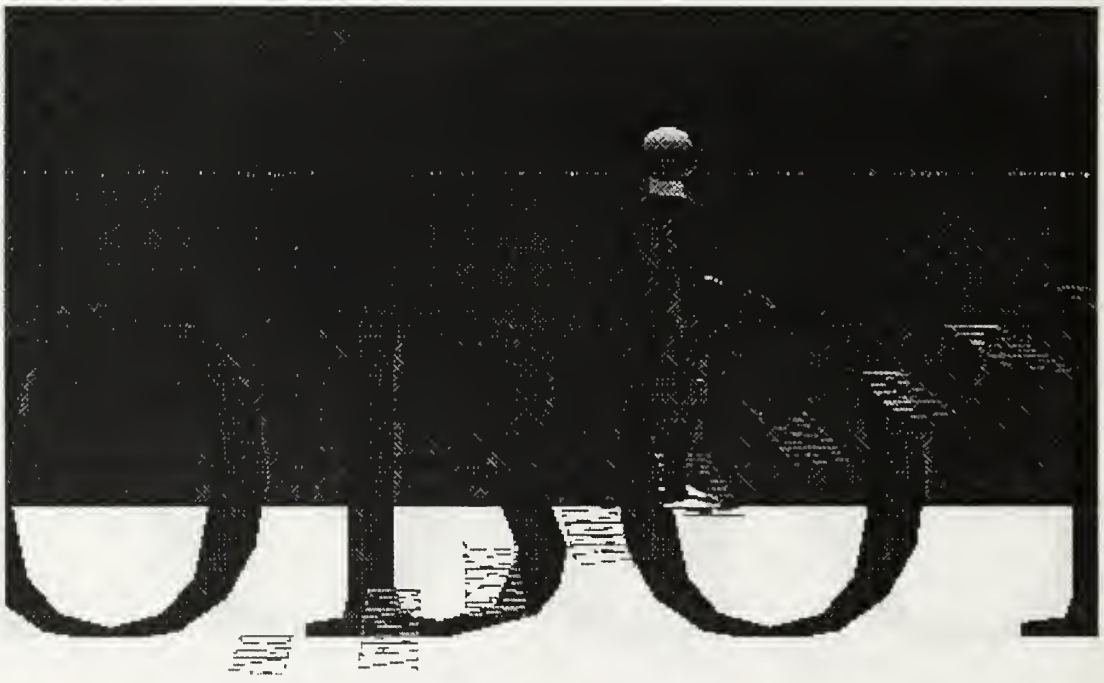

Figure 48: Upward Stepping (5)

VI. SUMMARY AND CONCLUSIONS

A. SUMMARY

The increasing demand for realistic 3D Virtual Environments requires more research on modeling human motion. There exist two main approaches to this problem: kinematic and dynamic modeling. A detailed review of previous work on kinematic modeling, dynamic modeling and control of graphical representation of human body motion is presented in this thesis. Kinematics uses only the geometric constraints of the model which results in less complicated models. However, for some situations, kinematic modeling may not be sufficient to simulate human motion realistically. Dynamics introduces additional properties of human limb segments to the simulation, such as mass and moment of inertia. These additional properties allow a more realistic simulation. The cost of this realism is an increase in computational complexity. The goal of the research of this thesis is to develop a realistic looking human model, while considering the complexity of the simulation according to the response time based on the computational power of current graphics hardware.

To be able to create a more realistic, real time, and computationally efficient human model, a simple dynamic model which was proposed, but not fully developed in [MCGH79] was implemented. The model is a two link planar inverted pendulum which is made of a rigid body with mass and a supporting massless leg. Two different methods, generalized coordinates with Lagrangian, and the Newton-Euler, were used to derive the dynamic equations of the simulations. The implementation results verified models against each other and related models in the literature [GUBI74]. The problem of determining the gain parameters for limb segment control torques is also investigated. Since the degree of the characteristic equation is four, an experimental solution is presented, instead of an analytic one. Another result is the difference between the running times of the simulations.

The simulation which uses the Newton-Euler method takes 20% longer than the Lagrangian version, because of matrix inversions. It is also observed that the Lagrangian differential equations are simpler to use in deriving gain values for the system in comparison to the Newton-Euler version of the problem. However, the Lagrangian method would be hard to use in a more complex model with higher degrees of freedom. For such cases, the recursive Articulated Body algorithm [MCMI95] which has O(n) complexity needs to be used.

On the other hand, a detailed kinematic human model which contains 14 body parts was also developed. The simulation is based on joints angle instead of the Euler angles which used in previous work. IRIS Performer^{TM} API, which interfaces to both IRIS GL^{TM} and OpenGL. IRIS Performer^{TM} and provides a hierarchy among Dynamic Coordinate Systems (DCS), was used to implement articulated chain structures easily.

B. CONCLUSION AND FUTURE RESEARCH

This thesis covers the basics for simplified dynamic human motion simulation. The model which is introduced in this thesis has only the torso with mass. An advanced version of this simulation may consider leg mass as well. The existing model is planar. Another step can be to investigate a 3D version of the model. The "Rigid-body" class could continue to be used for this purpose since provides support for simulation of body parts in 3D. Multilink more complex dynamic models also need to be investigated by considering different solution methods such as Newton-Euler, iterative use of inverse dynamics [KOOZ83], and the Articulated Body Algorithm [MCMI95]. Accuracy and time response of the system should be taken into account, while the complexity of the model is increased.

Another issue is to integrate the existing kinematic and dynamic models. This integration may result a more realistic human model than the developed kinematic model. It will be computationally cheaper than a 14 link human dynamic model.

This thesis investigated posture control of a human model. Stepping is another important subject in human motion simulation. An efficient automatic stepping algorithm needs to be to investigated for a dynamically simulated human model. Active (dynamic) balancing should be considered while studying automatic stepping control. Advance features in further research in stepping is speed and direction control, and rough terrain locomotion for the walking human model. For direction and speed control implementations, user-computer interaction will become important. A speech recognition interface could be taken in the consideration as a realistic solution [DEVI96].

VRML is a fast growing 3D modeling language. It is possible to execute the needed computation by using Java Classes which are integrated to VRML nodes. Implementing the simulations by using VRML and Java Scripting would provide platform independency which would allow the simulations runable in other than graphic workstation environments. The author hopes that the work of this thesis will provide a foundation for continuing research in some of the above topics.

APPENDIX A: DYNAMIC SIMULATION SOFTWARE

COMMON FILES IN ALL DYNAMIC SIMULATIONS

```
;; ***********************************************************************
;; File         : robot-kinematics.cl
;; Author       : Dr. R. McGhee
;;              : Naval Postgraduate School
;;              : Monterey, CA 93943
;; Summary      : This file contains various matrix and vector operation
;;              : functions.
;; ***********************************************************************

(defun transpose (matrix)           ;A matrix is a list of row vectors.
  (cond ((null (cdr matrix)) (mapcar 'list (car matrix)))
             (t (mapcar 'cons (car matrix) (transpose (cdr matrix))))))

(defun dot-product (vector-1 vector-2)  ;A vector is a list of numerical atoms.
  (apply '+ (mapcar '* vector-1 vector-2)))

(defun vector-magnitude (vector) (sqrt (dot-product vector vector)))

(defun post-multiply (matrix vector)
  (cond ((null (rest matrix)) (list (dot-product (first matrix) vector)))
             (t (cons (dot-product (first matrix) vector)
                 (post-multiply (rest matrix) vector)))))

(defun pre-multiply (vector matrix)
  (post-multiply (transpose matrix) vector))

(defun matrix-multiply (A B)      ;A and B are conformable matrices.
  (cond ((null (cdr A)) (list (pre-multiply (car A) B)))
             (t (cons (pre-multiply (car A) B) (matrix-multiply (cdr A) B)))))

(defun chain-multiply (L)        ;L is a list of names of conformable matrices.
  (cond ((null (cddr L)) (matrix-multiply (eval (car L)) (eval (cadr L))))
             (t (matrix-multiply (eval (car L)) (chain-multiply (cdr L))))))

(defun cycle-left (matrix) (mapcar 'row-cycle-left matrix))

(defun row-cycle-left (row) (append (cdr row) (list (car row))))

(defun cycle-up (matrix) (append (cdr matrix) (list (car matrix))))

(defun unit-vector (one-column length)         ;Column count starts at 1.
  (do ((n length (1- n))
       (vector nil (cons (cond ((= one-column n) 1) (t 0)) vector)))
      ((zerop n) vector)))



(defun unit-matrix (size)
  (do ((row-number size (1- row-number))
```

```
       (I nil (cons (unit-vector row-number size) I)))
      ((zerop row-number) I)))

(defun concat-matrix (A B)   ;A and B are matrices with equal number of rows.
  (cond ((null A) B)
            (t (cons (append (car A) (car B)) (concat-matrix (cdr A)
                                                             (cdr B))))))

(defun augment (matrix)
  (concat-matrix matrix (unit-matrix (length matrix))))

(defun normalize-row (row) (scalar-multiply (/ 1.0 (car row)) row))

(defun scalar-multiply (scalar vector)
  (cond ((null vector) nil)
            (t (cons (* scalar (car vector))
                (scalar-multiply scalar (cdr vector))))))

(defun solve-first-column (matrix)        ;Reduces first column to (1 0 ... 0).
  (do* ((remaining-row-list matrix (rest remaining-row-list))
        (first-row (normalize-row (first matrix)))
        (answer (list first-row)
                (cons (vector-add (first remaining-row-list)
                                  (scalar-multiply (- (caar remaining-row-list))
                                                   first-row))
                      answer)))
       ((null (rest remaining-row-list)) (reverse answer))))

(defun vector-add (vector-1 vector-2) (mapcar '+ vector-1 vector-2))

(defun vector-subtract (vector-1 vector-2) (mapcar '- vector-1 vector-2))

(defun first-square (matrix)  ;Returns leftmost square matrix from argument.
  (do ((size (length matrix))
       (remainder matrix (rest remainder))
       (answer nil (cons (firstn size (first remainder)) answer)))
      ((null remainder) (reverse answer))))

(defun firstn (n list)
  (cond ((zerop n) nil)
        (t (cons (first list) (firstn (1- n) (rest list))))))

(defun max-car-firstn (n list)
  (append (max-car-first (firstn n list)) (nthcdr n list)))

(defun matrix-inverse (M)
  (do ((M1 (max-car-first (augment M))
           (cond ((null M1) nil)       ;Abort for singular matrix.
                 (t (max-car-firstn n (cycle-left (cycle-up M1))))))
       (n (1- (length M)) (1- n)))
      ((or (minusp n) (null M1)) (cond ((null M1) nil) (t (first-square M1))))
      (setq M1 (cond ((zerop (caar M1)) nil) (t (solve-first-column M1))))))

(defun max-car-first (L)   ;L is a list of lists. This function finds list with
  (cond ((null (cdr L)) L) ;largest car and moves it to head of list of lists.
```

```
        (t (if (> (abs (caar L)) (abs (caar (max-car-first (cdr L))))) L
               (append (max-car-first (cdr L)) (list (car L)))))))

(defun dh-matrix (cosrotate sinrotate costwist sintwist length translate)
  (list (list cosrotate (- (* costwist sinrotate))
                        (* sintwist sinrotate) (* length cosrotate))
              (list sinrotate (* costwist cosrotate)
                        (- (* sintwist cosrotate)) (* length sinrotate))
              (list 0. sintwist costwist translate) (list 0. 0. 0. 1.)))

(defun homogeneous-transform (azimuth elevation roll x y z)
  (let ((spsi (sin azimuth)) (cpsi (cos azimuth)) (sth (sin elevation))
        (cth (cos elevation)) (sphi (sin roll)) (cphi (cos roll)))
       (list (list (* cpsi cth) (- (* cpsi sth sphi) (* spsi cphi))
                   (+ (* cpsi sth cphi) (* spsi sphi)) x)
                  (list (* spsi cth) (+ (* cpsi cphi) (* spsi sth sphi))
                        (- (* spsi sth cphi) (* cpsi sphi)) y)
             (list (- sth) (* cth sphi) (* cth cphi) z)
             (list 0. 0. 0. 1.))))

(defun inverse-H (H)          ;H is a 4x4 homogeneous transformation matrix.
  (let* ((minus-P (list (- (fourth (first H)))
                        (- (fourth (second H)))
                        (- (fourth (third H)))))
         (inverse-R (transpose (first-square (reverse (rest (reverse H))))))
         (inverse-P (post-multiply inverse-R minus-P)))
        (append (concat-matrix inverse-R (transpose (list inverse-P)))
                (list (list 0 0 0 1)))))

(defun rotation-matrix (azimuth elevation roll)
  (let ((spsi (sin azimuth)) (cpsi (cos azimuth)) (sth (sin elevation))
        (cth (cos elevation)) (sphi (sin roll)) (cphi (cos roll)))
       (list (list (* cpsi cth) (- (* cpsi sth sphi) (* spsi cphi))
                   (+ (* cpsi sth cphi) (* spsi sphi)))
                  (list (* spsi cth) (+ (* cpsi cphi) (* spsi sth sphi))
                        (- (* spsi sth cphi) (* cpsi sphi)))
             (list (- sth) (* cth sphi) (* cth cphi)))))

(defun body-rate-to-euler-rate-matrix (azimuth elevation roll)
  (let ((sth (sin elevation)) (cth (cos elevation)) (tth (tan elevation))
        (sphi (sin roll)) (cphi (cos roll)))
       (list (list 1 (* tth sphi) (* tth cphi))
             (list 0 cphi (- sphi))
             (list 0 (/ sphi cth) (/ cphi cth)))))
```

```
;; ***************************************************************************
;; File         : harmonic-equation.cl
;; Author       : Dr. R. McGhee
;;              : Naval Postgraduate School
;;              : Monterey, CA 93943
;; Summary      : This file contains functions to create a window and execute
;;              : drawing operations on the window.
;; ***************************************************************************
(require :xcw)
(cw:initialize-common-windows)

(defun make-window ()
  (cw:make-window-stream :borders 5
                         :left 10
                         :bottom 280
                         :width 600
                         :height 600
                         :title "Harmonic Equation"
                         :activate-p t))  ;Make window visible.

(defun scale-point-coordinates (x-y-list enlargement-factor)
  (let ((x (first x-y-list)) (y (second x-y-list)))
       (list (+ 50 (round (* enlargement-factor x)))
             (+ 225 (round (* enlargement-factor y))))))

(defun draw-coordinate-axes (window)
  (cw:draw-line window (cw:make-position :x 20 :y 225)
                       (cw:make-position :x 570 :y 225)
                       :brush-width 2)
  (cw:draw-line window (cw:make-position :x 50 :y 20)
                       (cw:make-position :x 50 :y 560)
                       :brush-width 2))

(defun draw-line-in-window (window enlargement-factor start end)
  (let ((scaled-start (scale-point-coordinates start enlargement-factor))
        (scaled-end (scale-point-coordinates end enlargement-factor)))
       (cw:draw-line window
         (cw:make-position :x (first scaled-start) :y (second scaled-start))
         (cw:make-position :x (first scaled-end) :y (second scaled-end)))))
```

SINGLE LINK INVERTED PENDULUM SIMULATION WITH NEWTON-EULER

```
;; ***************************************************************************
;; File         : euler-angle-rigid-body.cl
;; Author       : Dr. R. McGhee
;;              : Naval Postgraduate School
;;              : Monterey, CA 93943
;; Modified by  : Mehmet Bediz
;; Summary      : This file contains rigid body class which is implemented
;;              : by Dr. R. McGhee. The function "test-rigid-body-forces-and-
;;              : torques-three-with-M" and the other functions called by this
;;              : function are implemented by Mehmet Bediz.
;; ***************************************************************************

(defclass rigid-body
    ()
  ((posture                ;The vector (xe ye ze phi theta psi).
    :initform '(0 0 0 0 0 0)
    :initarg :posture
    :accessor posture)
   (posture-rate ;The vector (xe-dot ye-dot ze-dot phi-dot theta-dot psi-dot).
    :initarg :posture-rate
    :accessor posture-rate)
   (velocity               ;The six-vector (u v w p q r) in body coordinates.
    :initform '(0 0 0  0 0 0)
    :initarg :velocity
    :accessor velocity)
   (velocity-growth-rate  ;The vector (u-dot v-dot w-dot p-dot q-dot r-dot).
    :accessor velocity-growth-rate)
   (forces-and-torques    ;The vector (Fx Fy Fz L M N) in body coordinates.
    ;:initform (list 0 0 (- *gravity*) 0 0 0)
    :initform (list 0 0 0 0 0 0)
    :accessor forces-and-torques)
   (moments-of-inertia    ;The vector (Ix Iy Iz) in principal axis coordinates.
    :initform '(1 900 1)
    :initarg :moments-of-inertia
    :accessor moments-of-inertia)
   (mass
    :initform 100
    :initarg :mass
    :accessor mass)
   (node-list   ;(x y z 1) in body coord for each node. Starts with (0 0 0 1).
    :initform '((0 0 0 1) (-0.5 -0.5 0.5 1) (0.5 -0.5 0.5 1)(0.5 -0.5 -0.5 1)
                          (-0.5 -0.5 -0.5 1)(-0.5 0.5 0.5 1)(0.5  0.5 0.5 1)
                          (0.5  0.5 -0.5 1)(-0.5  0.5 -0.5 1)(0 0 0 1)
                          (0 0 3 1))
    :initarg :node-list
    :accessor node-list)
   (polygon-list
    :initform '((1 5 8 4)(5 6 7 8)(6 2 3 7)(2 1 4 3)(9 10))
    :initarg :polygon-list
    :accessor polygon-list)
   (transformed-node-list  ;(x y z 1) in earth coord for each node in node-list.
    :initform '((0 0 0 1) (-2 -2 0 1) (2 -2 0 1)(2 -2 -30 1)(-2 -2 -30 1)
                          (-2  2 0 1) (2  2 0 1)(2  2 -30 1)(-2  2 -30 1))
```

```
    :accessor transformed-node-list)
   (H-matrix
    :initform (unit-matrix 4)
    :accessor H-matrix)
   (time-stamp
    :accessor time-stamp)))

(defmethod initialize ((body rigid-body))
  (setf (transformed-node-list body) (node-list body))
  (setf (velocity-growth-rate body) (update-velocity-growth-rate body))
  (setf (posture-rate body) (earth-velocity body))
  (setf (time-stamp body) (get-internal-real-time))))

(defmethod set-transformed-node-list-z((body rigid-body) z)
   (setf (third(fourth (transformed-node-list body)))(- z))
   (setf (third(fifth (transformed-node-list body))) (- z))
   (setf (third(eighth (transformed-node-list body))) (- z))
   (setf (third(ninth (transformed-node-list body))) (- z))
  (transformed-node-list body)))

(defun transform (obj)
   (list 0 (first obj)(second obj)(third obj)))

(defun reverse-transform (obj)
   (list (second obj)(third obj)(fourth obj) 1))

(defmethod move ((body rigid-body) azimuth elevation roll x y z)
  (setf (posture body) (list x y z roll elevation azimuth))
  (setf (H-matrix body)
    (homogeneous-transform azimuth elevation roll x y z))
  (transform-node-list body))

(defmethod get-delta-t ((body rigid-body)) 0.02)
;  (let* ((new-time (get-internal-real-time))
;         (delta-t (/ (- new-time (time-stamp body)) 1000)))
;   (setf (time-stamp body) new-time)
;   delta-t))

(defmethod update-rigid-body ((body rigid-body))     ;Euler integration.
  (let* ((delta-t (get-delta-t body)))
       (update-posture body delta-t); EULER
       (update-velocity body delta-t); EULER
       (setf (H-matrix body) (homogeneous-transform (sixth (posture body))
          (fifth (posture body)) (fourth (posture body)) (first (posture body))
          (second (posture body)) (third (posture body))))
       (transform-node-list body)
       (update-velocity-growth-rate body)))

(defmethod update-velocity-growth-rate ((body rigid-body))
  (setf (velocity-growth-rate body) ;Assumes principal axis coordinates with
    (multiple-value-bind           ;origin at center of gravity of body.
      (Fx Fy Fz L M N   u v w p q r   Ix Iy Iz) ;Declares local variables.
      (values-list                              ;Values assigned.
        (append
          (forces-and-torques body) (velocity body) (moments-of-inertia body)))
```

```
      (list (+ (* v r) (* -1 w q) (/ Fx (mass body))
               (* *gravity* (first (third (H-matrix body)))))
            (+ (* w p) (* -1 u r) (/ Fy (mass body))
               (* *gravity* (second (third (H-matrix body)))))
            (+ (* u q) (* -1 v p) (/ Fz (mass body))
               (* *gravity* (third (third (H-matrix body)))))
            (/ (+ (* (- Iy Iz) q r) L) Ix)
            (/ (+ (* (- Iz Ix) r p) M) Iy)
            (/ (+ (* (- Ix Iy) p q) N) Iz)))))  ;Value returned.

(defmethod update-velocity ((body rigid-body) delta-t) ;Euler integration.
  (setf (velocity body)
    (vector-add (velocity body)
                (scalar-multiply delta-t (velocity-growth-rate body)))))

(defmethod update-posture ((body rigid-body) delta-t) ;Euler integration.
  (setf (posture-rate body) (earth-velocity body))
  (setf (posture body)
     (vector-add (posture body) (scalar-multiply delta-t (posture-rate body)))))

(defmethod transform-node-list ((body rigid-body))
  (setf (transformed-node-list body)
    (mapcar #'(lambda (node-location)
                (post-multiply (H-matrix body) node-location))
            (node-list body))))

(defconstant *gravity* 32.2185)

(defmethod earth-velocity ((body rigid-body))
  (let* ((linear-velocity (firstn 3 (velocity body)))
         (rotational-velocity (cdddr (velocity body)))
         (posture (posture body))
         (R-matrix (rotation-matrix (sixth posture) (fifth posture)
                                    (fourth posture)))
         (linear-earth-velocity (post-multiply R-matrix linear-velocity))
         (T-matrix (body-rate-to-euler-rate-matrix (sixth posture)
                     (fifth posture) (fourth posture)))
         (rotational-earth-velocity (post-multiply T-matrix
                                                   rotational-velocity)))
        (append linear-earth-velocity rotational-earth-velocity)))

(defun test-rigid-body ()
  (setf airplane-1 (make-instance 'rigid-body))
  (initialize airplane-1)
  (setf camera-1 (make-instance 'strobe-camera))
  (move camera-1 (- (/ pi 2)) 0  0 0 30 0 )
  (take-picture camera-1 airplane-1)
  (dotimes (i 20 'done) (update-rigid-body airplane-1))
  (take-picture camera-1 airplane-1))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;              - Inverted Pendulum (fixed leg length)
;              - state vector x = (theta theta-dot)
;              - state vector u = (M)
;                 - control torque at the pivot (M)
;              - Newton-Euler Method
;                 -  3x3 matrix inversion to calculate forces and moments
;                 -  RIGID Body class to compute accelerations
;              - Heun Integration
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod forces-earth-to-body (Fx-e Fy-e Fz-e theta)
  (let* ((forces-earth (list Fx-e Fy-e Fz-e))

         (R-matrix (rotation-matrix 0 (- theta) 0)) ;Since planar
         (forces-body (post-multiply R-matrix forces-earth)))

        forces-body))

(defmethod update-forces-and-torques-three-with-M ((body rigid-body) M)
   (let*  ((equations-solution-vector (solve-three-equation-system-with-M
                                                                    body M))
           (Fx-earth (second equations-solution-vector))
           (Fz-earth (third equations-solution-vector))
           (F-body  (forces-earth-to-body Fx-earth 0 Fz-earth
                                                  (fifth (posture body)))))
       (setf forces-and-torques (list (first F-body)
                                      0
                                      (third F-body)
                                      0
                                      (* (first equations-solution-vector)
                                   (second (moments-of-inertia body)))
                                      0))))

(defun solve-three-equation-system-with-M (body M)
   (setf  m       (mass body))
   (setf  l        3)
   (setf  I       (second (moments-of-inertia body)))
   (setf  theta   (fifth  (posture  body)))
   (setf  theta-dot   (fifth  (velocity body)))
   (setf  g    *gravity*)

(post-multiply (matrix-inverse (list (list I (* -1 l (cos theta))
                                             (* l (sin theta)))
                                     (list (* m  l (cos theta))    l      0 )
                         (list (* m l (sin theta))   0   -1 )))
                        (list  M
                               (* (sqr theta-dot) m l (sin theta))
                               (- (* m g) (* (sqr theta-dot) m l (cos theta))))))
(defun compute-M (body K-theta K-theta-dot)
   (setf  theta   (fifth  (posture  body)))
   (setf  theta-dot   (fifth  (velocity body)))
```

```
   (+ (* -1 K-theta theta) (* -1 K-theta-dot  theta-dot)))

(defun test-rigid-body-forces-and-torques-three-with-M (K-theta K-theta-dot)
  (setf inverted-pendulum (make-instance 'rigid-body))
  (initialize inverted-pendulum)
  (setf camera-1 (make-instance 'strobe-camera))
  (move camera-1 (deg-to-rad (- 90))(deg-to-rad 0)(deg-to-rad 0) 0 10 0)
  (move inverted-pendulum 0 -1 0 0 0 0)
  ;(take-picture camera-1 inverted-pendulum)

   (setf graph-1-window (make-window-graph-1))
   (draw-coordinate-axes graph-1-window)

   (setf old-theta 0.5)
  (do ((i 0 (+ i 1)))
       (( > i 1000)  'end)

   (setf (forces-and-torques inverted-pendulum)
              (update-forces-and-torques-three-with-M inverted-pendulum
                    (compute-M inverted-pendulum K-theta K-theta-dot)))
   (update-rigid-body inverted-pendulum)
   (cw:clear (camera-window camera-1))
   (take-picture camera-1 inverted-pendulum)
   (draw-line-in-window graph-1-window 80
                                (list
                                   (*  (get-delta-t inverted-pendulum) (- i 1))
                                   (* -1  old-theta))
                                (list
                                   (*   (get-delta-t inverted-pendulum) i)
                                   (* -1 (fifth (posture inverted-pendulum)))))

   (setf old-theta (fifth (posture inverted-pendulum)))))

(defun sqr(x)
   (* x x))

(defun make-window-graph-1 ()
  (cw:make-window-stream :borders 5
                         :left 0
                         :bottom 550
                         :width 450
                         :height 450
                         :title "Body Attitude   (theta)   NEWTON-EULER"
                         :activate-p t))  ;Make window visible.
```

```
;; ***************************************************************************
;; File         : load-euler-files.cl
;; Author       : Mehmet Bediz
;;              : Naval Postgraduate School
;;              : Monterey, CA 93943
;; Summary      : This file contains functions to load files and to test
;;              : the simulations.
;; ***************************************************************************

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;        Load files
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(load "harmonic-equation.cl")
(load "robot-kinematics.cl")
(load "euler-angle-rigid-body.cl")
(load "strobe-camera.cl")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;           Tests
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun graph_1 ()
    (solve-robot-attitude 0.01 1.0 10000 2000 109660 20000))

(defun graph_2 ()
    (solve-robot-altitude-z_z0 0.01 1.0 10000 2000 109660 20000))

(defun graph_3 ()
    (solve-robot-altitude-z 0.01 1.0 10000 2000 109660 20000))

(defun draw-converted-pendulum-test-real-parameters ()
      (draw-converted-pendulum 0.01 2.0 10000 2000 109660 20000))

(defun draw-converted-pendulum-test-zero-gain ()
      (draw-converted-pendulum 0.01 2.0  0      0     0     0))
```

```
;; ***************************************************************************
;; File         : strobe-camera.cl
;; Author       : Dr. R. McGhee
;;              : Naval Postgraduate School
;;              : Monterey, CA 93943
;; Summary      : This file contains strobe-camera class definitions.
;; ***************************************************************************

(require :xcw)
(cw:initialize-common-windows)

(defclass strobe-camera (rigid-body)
  ((focal-length
    :accessor focal-length
    :initform 6)
   (camera-window
    :accessor camera-window
    :initform (cw:make-window-stream :borders 5
                                     :left 500
                                     :bottom 500
                                     :width 650
                                     :height 650
                                     :title "strobe-camera-image"
                                     :activate-p t))
   (H-matrix
    :initform (homogeneous-transform .3 -.3 0 -300 -90 -90))
   (inverse-H-matrix
    :accessor inverse-H-matrix
    :initform (inverse-H (homogeneous-transform .3 -.3 0 -300 -90 -90)))
   (enlargement-factor
    :accessor enlargement-factor
    :initform 30)))

(defmethod move ((camera strobe-camera) azimuth elevation roll x y z)
  (setf (H-matrix camera) (homogeneous-transform azimuth elevation roll x y z))
  (setf (inverse-H-matrix camera) (inverse-H (H-matrix camera))))

(defmethod take-picture ((camera strobe-camera) (body rigid-body))
  (let ((camera-space-node-list (mapcar #'(lambda (node-location)
         (post-multiply (inverse-H-matrix camera) node-location))
                        (transformed-node-list body))))
    (dolist (polygon (polygon-list body))
      (clip-and-draw-polygon camera polygon camera-space-node-list))))

(defmethod clip-and-draw-polygon
  ((camera strobe-camera) polygon node-coord-list)
  (do* ((initial-point (nth (first polygon) node-coord-list))
        (from-point initial-point to-point)
        (remaining-nodes (rest polygon) (rest remaining-nodes))
        (to-point (nth (first remaining-nodes) node-coord-list)
                  (if (not (null (first remaining-nodes)))
                      (nth (first remaining-nodes) node-coord-list))))
       ((null to-point)
              (draw-clipped-projection camera from-point initial-point))
       (draw-clipped-projection camera from-point to-point)))
```

```
(defmethod draw-clipped-projection ((camera strobe-camera) from-point to-point)
  (cond ((and (<= (first from-point) (focal-length camera))
              (<= (first to-point) (focal-length camera))) nil)
        ((<= (first from-point) (focal-length camera))
         (draw-line-in-camera-window camera
           (perspective-transform camera (from-clip camera from-point to-point))
           (perspective-transform camera to-point)))
        ((<= (first to-point) (focal-length camera))
         (draw-line-in-camera-window camera
           (perspective-transform camera from-point)
           (perspective-transform camera (to-clip camera from-point to-point))))
        (t (draw-line-in-camera-window camera
           (perspective-transform camera from-point)
           (perspective-transform camera to-point)))))

(defmethod from-clip ((camera strobe-camera) from-point to-point)
  (let ((scale-factor (/ (- (focal-length camera) (first from-point))
                         (- (first to-point) (first from-point)))))
       (list (+ (first from-point)
                (* scale-factor (- (first to-point) (first from-point))))
             (+ (second from-point)
                (* scale-factor (- (second to-point) (second from-point))))
             (+ (third from-point)
                (* scale-factor (- (third to-point) (third from-point)))) 1)))

(defmethod to-clip ((camera strobe-camera) from-point to-point)
  (from-clip camera to-point from-point))

(defmethod draw-line-in-camera-window ((camera strobe-camera) start end)
  (cw:draw-line (camera-window camera)
                (cw:make-position :x (+ 150 (first start)) :y (+ 150 (second
start)))
                (cw:make-position :x (+ 150 (first end)) :y (+ 150 (second
end)))
                :brush-width 0))

(defmethod perspective-transform ((camera strobe-camera) point-in-camera-space)
  (let* ((enlargement-factor (enlargement-factor camera))
              (focal-length (focal-length camera))
              (x (first point-in-camera-space))  ;x axis is along optical axis
              (y (second point-in-camera-space)) ;y is out right side of camera
              (z (third point-in-camera-space))) ;z is out bottom of camera
             (list (+ (round (* enlargement-factor (/ (* focal-length y) x)))
                      150)        ;to right in camera window
              (+ 150 (round (* enlargement-factor (/ (* focal-length (- z)) x))
                            )))))  ;up in camera window

(defun test-camera (z theta) ;Produces top view of default rigid-body.
  (setf robot-leg (make-instance 'rigid-body))
  (initialize  robot-leg)
  (set-transformed-node-list-z robot-leg z)
  (set-transformed-node-list-theta robot-leg theta)
  (setf camera-1 (make-instance 'strobe-camera))
  (move camera-1 (deg-to-rad 0)(deg-to-rad 0)(deg-to-rad 0) -30 0 0)
```

```
  (take-picture camera-1 robot-leg))

(defun deg-to-rad (angle) (* .01745329251994329 angle))
```

TWO LINK INVERTED PENDULUM SIMULATION WITH LAGRANGIAN

```
;; ***************************************************************************
;; File         : euler-angle-rigid-body.cl
;; Author       : Dr. R. McGhee
;;              : Naval Postgraduate School
;;              : Monterey, CA 93943
;; Modified by  : Mehmet Bediz
;; Summary      : This file contains rigid body class which is implemented
;;              : by Dr. R. McGhee. The function "draw-inverted-pendulum"
;;              : and the other functions called by this
;;              : function are implemented by Mehmet Bediz.
;; ***************************************************************************

(defclass rigid-body
    ()
  ((posture                ;The vector (xe ye ze phi theta psi).
    :initform '(0 0 0 0 0 0)
    :initarg :posture
    :accessor posture)
   (posture-rate ;The vector (xe-dot ye-dot ze-dot phi-dot theta-dot psi-dot).
    :initarg :posture-rate
    :accessor posture-rate)
   (velocity               ;The six-vector (u v w p q r) in body coordinates.
    :initform '(0 0 0  0 0 0)
    :initarg :velocity
    :accessor velocity)
   (velocity-growth-rate  ;The vector (u-dot v-dot w-dot p-dot q-dot r-dot).
    :accessor velocity-growth-rate)
   (forces-and-torques    ;The vector (Fx Fy Fz L M N) in body coordinates.
    :initform (list 0 0 0 0 0 0)
    :accessor forces-and-torques)
   (moments-of-inertia    ;The vector (Ix Iy Iz) in principal axis coordinates.
    :initform '(1 100 1)
    :initarg :moments-of-inertia
    :accessor moments-of-inertia)
   (mass
    :initform 100
    :initarg :mass
    :accessor mass)
   (node-list    ;(x y z 1) in body coord for each node. Starts with (0 0 0 1).
    :initform '((0 0 0 1) (-0.5 -0.5 0.5 1) (0.5 -0.5 0.5 1)(0.5 -0.5 -0.5 1)
                          (-0.5 -0.5 -0.5 1)(-0.5  0.5 0.5 1) (0.5  0.5 0.5 1)
                          (0.5  0.5 -0.5 1)(-0.5  0.5 -0.5 1)(0 0 0 1)(0 0 3 1))
    :initarg :node-list
    :accessor node-list)
   (polygon-list
    :initform '((1 5 8 4)(5 6 7 8)(6 2 3 7)(2 1 4 3)(9 10))
    :initarg :polygon-list
    :accessor polygon-list)
   (transformed-node-list  ;(x y z 1) in earth coord for each node in node-list.
    :initform '((0 0 0 1) (-2 -2 0 1) (2 -2 0 1)(2 -2 -30 1)(-2 -2 -30 1)
                           (-2  2 0 1) (2  2 0 1)(2  2 -30 1)(-2  2 -30 1))
    :accessor transformed-node-list)
   (H-matrix
```

```
    :initform (unit-matrix 4)
    :accessor H-matrix)
   (time-stamp
    :accessor time-stamp)))

(defmethod initialize ((body rigid-body))
  (setf (transformed-node-list body) (node-list body))
  (setf (velocity-growth-rate body) (update-velocity-growth-rate body))
  (setf (posture-rate body) (earth-velocity body))
  (setf (time-stamp body) (get-internal-real-time))))

(defmethod set-transformed-node-list-z((body rigid-body) z)
   (setf (third(fourth (transformed-node-list body)))(- z))
   (setf (third(fifth (transformed-node-list body))) (- z))
   (setf (third(eighth (transformed-node-list body))) (- z))
   (setf (third(ninth (transformed-node-list body))) (- z))
   (transformed-node-list body)))

(defun transform (obj)
   (list 0 (first obj)(second obj)(third obj)))

(defun reverse-transform (obj)
   (list (second obj)(third obj)(fourth obj) 1))

(defmethod move ((body rigid-body) azimuth elevation roll x y z)
  (setf (posture body) (list x y z roll elevation azimuth))
  (setf (H-matrix body)
    (homogeneous-transform azimuth elevation roll x y z))
  (transform-node-list body))

(defmethod get-delta-t ((body rigid-body)) 0.005)

(defmethod forces-earth-to-body (Fx-e Fy-e Fz-e theta)
  (let* ((forces-earth (list Fx-e Fy-e Fz-e))

         (R-matrix (rotation-matrix 0 (- theta) 0))
         (forces-body (post-multiply R-matrix forces-earth)))

        forces-body))

(defmethod update-rigid-body ((body rigid-body))     ;Euler integration.
  (let* ((delta-t (get-delta-t body)))
       (update-posture body delta-t); EULER
       (update-velocity body delta-t); EULER
       (setf (H-matrix body) (homogeneous-transform (sixth (posture body))
          (fifth (posture body)) (fourth (posture body)) (first (posture body))
          (second (posture body)) (third (posture body))))
       (transform-node-list body)

       (update-velocity-growth-rate body)))

(defmethod update-velocity-growth-rate ((body rigid-body))
  (setf (velocity-growth-rate body) ;Assumes principal axis coordinates with
    (multiple-value-bind           ;origin at center of gravity of body.
      (Fx Fy Fz L M N   u v w p q r   Ix Iy Iz) ;Declares local variables.
```

```
      (values-list                                ;Values assigned.
        (append
          (forces-and-torques body) (velocity body) (moments-of-inertia body)))
      (list (+ (* v r) (* -1 w q) (/ Fx (mass body))
               (* *gravity* (first (third (H-matrix body)))))
            (+ (* w p) (* -1 u r) (/ Fy (mass body))
               (* *gravity* (second (third (H-matrix body)))))
            (+ (* u q) (* -1 v p) (/ Fz (mass body))
               (* *gravity* (third (third (H-matrix body)))))
            (/ (+ (* (- Iy Iz) q r) L) Ix)
            (/ (+ (* (- Iz Ix) r p) M) Iy)
            (/ (+ (* (- Ix Iy) p q) N) Iz)))))  ;Value returned.

(defmethod update-velocity ((body rigid-body) delta-t) ;Euler integration.
  (setf (velocity body)
    (vector-add (velocity body)
                (scalar-multiply delta-t (velocity-growth-rate body)))))

(defmethod update-posture ((body rigid-body) delta-t) ;Euler integration.
  (setf (posture-rate body) (earth-velocity body))
  (setf (posture body)
     (vector-add (posture body) (scalar-multiply delta-t (posture-rate body)))))

(defmethod transform-node-list ((body rigid-body))
  (setf (transformed-node-list body)
    (mapcar #'(lambda (node-location)
                (post-multiply (H-matrix body) node-location))
            (node-list body))))

(defconstant *gravity* 32.2185)

(defmethod earth-velocity ((body rigid-body))
  (let* ((linear-velocity (firstn 3 (velocity body)))
         (rotational-velocity (cdddr (velocity body)))
         (posture (posture body))
         (R-matrix (rotation-matrix (sixth posture) (fifth posture)
                                    (fourth posture)))
         (linear-earth-velocity (post-multiply R-matrix linear-velocity))
         (T-matrix (body-rate-to-euler-rate-matrix (sixth posture)
                     (fifth posture) (fourth posture)))
         (rotational-earth-velocity (post-multiply T-matrix
                                                   rotational-velocity)))
        (append linear-earth-velocity rotational-earth-velocity)))

(defun test-rigid-body ()
  (setf airplane-1 (make-instance 'rigid-body))
  (initialize airplane-1)
  (setf camera-1 (make-instance 'strobe-camera))
  (move camera-1 (- (/ pi 2)) 0  0 0 30 0 )
  (take-picture camera-1 airplane-1)
  (dotimes (i 20 'done) (update-rigid-body airplane-1))
  (take-picture camera-1 airplane-1))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;               - Two Link Inverted Pendulum
;               - Generalized Coordinates with Lagrangian Method
;               - Massless leg and a single rigid body (fixed leg length)
;               - State vector x = (theta-1 theta-1-dot theta-2 theta-2-dot)
;               - State vector u = (M-1 M-2)
;                  - control torque at the ankle (M-1)
;                  - control torque at the hip (M-2)
;               - Heun Integration
;               - By Mehmet Bediz
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   (setf  l       0.5)
   (setf  r       3)
   (setf time-step 0.01)

(defun euler-step-constant-leg (state-vector M-1 M-2)
  (mapcar '+ state-vector (scalar-multiply time-step
                              (derivative-state-vector
                               state-vector M-1 M-2))))

(defun heun-step-constant-leg (x K-theta-1 K-theta-1-dot
                                 K-theta-2 K-theta-2-dot)
   (setf  M-1 (compute-M-1 x K-theta-1 K-theta-1-dot
                                K-theta-2 K-theta-2-dot))
   (setf  M-2 (compute-M-2 x K-theta-1 K-theta-1-dot
                                K-theta-2 K-theta-2-dot))
  (mapcar '+ x (scalar-multiply (* time-step .5)
    (vector-add (derivative-state-vector x M-1 M-2)
                (derivative-state-vector
                     (euler-step-constant-leg x M-1 M-2) M-1 M-2)))))

(defun derivative-state-vector (state-vector  M-1 M-2)
           (list (second state-vector)
                 (compute-theta-1-double-dot state-vector  M-1 M-2)
                 (fourth state-vector)
                 (compute-theta-2-double-dot state-vector  M-1 M-2)))

(defun compute-theta-1-double-dot (state-vector M-1 M-2) ; Eq 3.51
   (setf  m       100)
   (setf  l       0.5)
   (setf  r       3)
   (setf  I       100)
   (setf  theta-1       (first  state-vector))
   (setf  theta-1-dot   (second state-vector))
   (setf  theta-2       (third  state-vector))
   (setf  theta-2-dot   (fourth state-vector))
   (setf  g    32.2)

    ( / (+ (* l l m m r g theta-2)
           (* -1 m g r (+ I (* m l l)) theta-1)
           (* -1 (+ I (* m l l)) M-1)
           (* (+ (* l m r) I (* m l l)) M-2)
        )
      (* -1 I m r r)))
```

```
(defun compute-theta-2-double-dot (state-vector M-1 M-2) ; Eq 3.54
   (setf  m       100)
   (setf  l       0.5)
   (setf  r       3)
   (setf  I       100)
   (setf  theta-1       (first  state-vector))
   (setf  theta-1-dot   (second state-vector))
   (setf  theta-2       (third  state-vector))
   (setf  theta-2-dot   (fourth state-vector))
   (setf  g    32.2)

    ( / (+ (* m g l r theta-2)
           (* -1 m g l r theta-1)
           (* -l l M-1)
           (* (+ r l) M-2)
       )
      (* r I)
    )
)

(defun compute-M-1 (state-vector K-theta-1 K-theta-1-dot
                                 K-theta-2 K-theta-2-dot)
  (setf theta-1 (first state-vector))
  (setf theta-1-dot (second state-vector))
   (+ (* -1 K-theta-1 theta-1)(* -1 K-theta-1-dot theta-1-dot)))

(defun compute-M-2 (state-vector K-theta-1 K-theta-1-dot
                                 K-theta-2 K-theta-2-dot)
  (setf theta-2 (third state-vector))
  (setf theta-2-dot (fourth state-vector))
   (+ (* -1 K-theta-2 theta-2)(* -1 K-theta-2-dot theta-2-dot)))

(defun compute-y (state-vector)
   (setf  theta-1       (first  state-vector))
   (setf  theta-1-dot   (second state-vector))
   (setf  theta-2       (third  state-vector))
   (setf  theta-2-dot   (fourth state-vector))
   (setf  l       0.5)
   (setf  r       3)
   (+ (* r (sin theta-1))
      (* l (sin theta-2))))

(defun compute-z (state-vector)
   (setf  theta-1       (first  state-vector))
   (setf  theta-1-dot   (second state-vector))
   (setf  theta-2       (third  state-vector))
   (setf  theta-2-dot   (fourth state-vector))
   (setf  l       0.5)
   (setf  r       3)
   (+ (* r (cos theta-1))
      (* l (cos theta-2))))


(defun draw-inverted-pendulum (halt-time K-theta-1 K-theta-1-dot
                                         K-theta-2 K-theta-2-dot)
```

```
    (setf robot-body (make-instance 'rigid-body))
    (initialize  robot-body)

    (setf robot-leg (make-instance 'rigid-body))
    (initialize  robot-leg)


    (setf camera-1 (make-instance 'strobe-camera))
    (move camera-1 (deg-to-rad 0)(deg-to-rad 0)(deg-to-rad 0) -10 0 0)

    (setf graph-1-window (make-window-graph-1))
    (draw-coordinate-axes graph-1-window)

    (setf graph-2-window (make-window-graph-2))
    (draw-coordinate-axes graph-2-window)

    (do ((x '(0.5 0 1.0 0) (heun-step-constant-leg x K-theta-1 K-theta-1-dot
                                                   K-theta-2 K-theta-2-dot))
        (old-x '(0.5 0 1.0 0) x)
        (time 0 (+ time 1)))
      ((> time halt-time) 'done)

      (move  robot-body (deg-to-rad 0)(deg-to-rad 0) (third x)
                        0            (compute-y x) (* -1(compute-z x)))

      (setf (transformed-node-list robot-leg) (list (list 0 0 0 1)
                                                    (list 0 (* r (sin (first x)))
                                                    (* -1  r (cos (first x))) 1)
                                                    (list 0 0 0 1)))
      (setf (polygon-list robot-leg) (list (list 2 1)))

      (cw:clear (camera-window camera-1))
      (take-picture camera-1 robot-body)
      (take-picture camera-1 robot-leg)
      (draw-line-in-window graph-1-window 80 (list (* time-step (- time 1))
                                                   (first old-x))
                                             (list (* time-step time)
                                                   (first x)))
      (draw-line-in-window graph-2-window 80 (list (* time-step (- time 1))
                                                   (third old-x))
                                             (list (* time-step time)
                                                   (third x)))))

(defun make-window-graph-1 ()
  (cw:make-window-stream :borders 5
                         :left 0
                         :bottom 50
                         :width 450
                         :height 450
                         :title "Leg Angle   (theta-1) LAGRANGIAN"
                         :activate-p t))  ;Make window visible.

(defun make-window-graph-2 ()
  (cw:make-window-stream :borders 5
                         :left 0
```

```
:bottom 550
:width 450
:height 450
:title "Body Attitude   (theta-2) LAGRANGIAN"
:activate-p t))  ;Make window visible.
```

```
;; ***************************************************************************
;; File         : load-euler-files.cl
;; Author       : Mehmet Bediz
;;              : Naval Postgraduate School
;;              : Monterey, CA 93943
;; Summary      : This file contains functions to load files.
;; ***************************************************************************

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;        Load files
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(load "harmonic-equation.cl")
(load "robot-kinematics.cl")
(load "euler-angle-rigid-body.cl")
```

```
;; ***********************************************************************
;; File        : strobe-camera.cl
;; Author      : Dr. R. McGhee
;;             : Naval Postgraduate School
;;             : Monterey, CA 93943
;; Summary     : This file contains strobe-camera class definitions.
;; ***********************************************************************

(require :xcw)
(cw:initialize-common-windows)

(defclass strobe-camera (rigid-body)
  ((focal-length
    :accessor focal-length
    :initform 6)
   (camera-window
    :accessor camera-window
    :initform (cw:make-window-stream :borders 5
                                     :left 470
                                  :bottom 500
                                  :width 650
                                  :height 650
                                  :title "Two Link Inverted Pendulum
                                            (Constant Length Massless Leg)
                                             LAGRANGIAN"
                                   :activate-p t))
   (H-matrix
    :initform (homogeneous-transform .3 -.3 0 -300 -90 -90))
   (inverse-H-matrix
    :accessor inverse-H-matrix
    :initform (inverse-H (homogeneous-transform .3 -.3 0 -300 -90 -90)))
   (enlargement-factor
    :accessor enlargement-factor
    :initform 30)))

(defmethod move ((camera strobe-camera) azimuth elevation roll x y z)
  (setf (H-matrix camera) (homogeneous-transform azimuth elevation roll x y z))
  (setf (inverse-H-matrix camera) (inverse-H (H-matrix camera))))

(defmethod take-picture ((camera strobe-camera) (body rigid-body))
  (let ((camera-space-node-list (mapcar #'(lambda (node-location)
         (post-multiply (inverse-H-matrix camera) node-location))
                        (transformed-node-list body))))
    (dolist (polygon (polygon-list body))
      (clip-and-draw-polygon camera polygon camera-space-node-list))))

(defmethod clip-and-draw-polygon
  ((camera strobe-camera) polygon node-coord-list)
  (do* ((initial-point (nth (first polygon) node-coord-list))
        (from-point initial-point to-point)
        (remaining-nodes (rest polygon) (rest remaining-nodes))
        (to-point (nth (first remaining-nodes) node-coord-list)
                  (if (not (null (first remaining-nodes)))
                      (nth (first remaining-nodes) node-coord-list))))
       ((null to-point)
```

```
            (draw-clipped-projection camera from-point initial-point))
       (draw-clipped-projection camera from-point to-point)))

(defmethod draw-clipped-projection ((camera strobe-camera) from-point to-point)
  (cond ((and (<= (first from-point) (focal-length camera))
              (<= (first to-point) (focal-length camera))) nil)
        ((<= (first from-point) (focal-length camera))
         (draw-line-in-camera-window camera
           (perspective-transform camera (from-clip camera from-point to-point))
           (perspective-transform camera to-point)))
        ((<= (first to-point) (focal-length camera))
         (draw-line-in-camera-window camera
           (perspective-transform camera from-point)
           (perspective-transform camera (to-clip camera from-point to-point))))
        (t (draw-line-in-camera-window camera
           (perspective-transform camera from-point)
           (perspective-transform camera to-point)))))

(defmethod from-clip ((camera strobe-camera) from-point to-point)
  (let ((scale-factor (/ (- (focal-length camera) (first from-point))
                         (- (first to-point) (first from-point)))))
       (list (+ (first from-point)
                (* scale-factor (- (first to-point) (first from-point))))
             (+ (second from-point)
                (* scale-factor (- (second to-point) (second from-point))))
             (+ (third from-point)
                (* scale-factor (- (third to-point) (third from-point)))) 1)))

(defmethod to-clip ((camera strobe-camera) from-point to-point)
  (from-clip camera to-point from-point))

(defmethod draw-line-in-camera-window ((camera strobe-camera) start end)
  (cw:draw-line (camera-window camera)
                (cw:make-position :x (+ 150 (first start))
                                  :y (+ 150 (second start)))
                (cw:make-position :x (+ 150 (first end))
                                  :y (+ 150 (second end)))
                                  :brush-width 0))

(defmethod perspective-transform ((camera strobe-camera) point-in-camera-space)
  (let* ((enlargement-factor (enlargement-factor camera))
              (focal-length (focal-length camera))
              (x (first point-in-camera-space))  ;x axis is along optical axis
              (y (second point-in-camera-space)) ;y is out right side of camera
              (z (third point-in-camera-space))) ;z is out bottom of camera
             (list (+ (round (* enlargement-factor (/ (* focal-length y) x)))
                              150)  ;to right in camera window
              (+ 150 (round (* enlargement-factor (/ (* focal-length (- z)) x))
                               )))))  ;up in camera window

(defun test-camera (z theta) ;Produces top view of default rigid-body.
  (setf robot-leg (make-instance 'rigid-body))
  (initialize  robot-leg)
  (set-transformed-node-list-z robot-leg z)
  (set-transformed-node-list-theta robot-leg theta)
```

```
  (setf camera-1 (make-instance 'strobe-camera))
  (move camera-1 (deg-to-rad 0)(deg-to-rad 0)(deg-to-rad 0) -30 0 0)
  (take-picture camera-1 robot-leg))

(defun deg-to-rad (angle) (* .01745329251994329 angle))

(defun animation ()

    (do ((theta-loop 0 (+ theta-loop 1)))
    ((> theta-loop 90) 'end)
    (test-camera 30 theta-loop)))
```

TWO LINK INVERTED PENDULUM SIMULATION WITH NEWTON-EULER

```
;; ***************************************************************************
;; File         : euler-angle-rigid-body.cl
;; Author       : Dr. R. McGhee
;;              : Naval Postgraduate School
;;              : Monterey, CA 93943
;; Modified by : Mehmet Bediz
;; Summary      : This file contains rigid body class which is implemented
;;              : by Dr. R. McGhee. The function "draw-inverted-pendulum"
;;              : and the other functions called by this
;;              : function are implemented by Mehmet Bediz.
;; ***************************************************************************

(defclass rigid-body
    ()
  ((posture                ;The vector (xe ye ze phi theta psi).
    :initform '(0 0 0 0 0 0)
    :initarg :posture
    :accessor posture)
   (posture-rate ;The vector (xe-dot ye-dot ze-dot phi-dot theta-dot psi-dot).
    :initarg :posture-rate
    :accessor posture-rate)
   (velocity               ;The six-vector (u v w p q r) in body coordinates.
    :initform '(0 0 0  0 0 0)
    :initarg :velocity
    :accessor velocity)
   (velocity-growth-rate  ;The vector (u-dot v-dot w-dot p-dot q-dot r-dot).
    :accessor velocity-growth-rate)
   (forces-and-torques    ;The vector (Fx Fy Fz L M N) in body coordinates.
    :initform (list 0 0 0 0 0 0)
    :accessor forces-and-torques)
   (moments-of-inertia    ;The vector (Ix Iy Iz) in principal axis coordinates.
    :initform '(1 100 1)
    :initarg :moments-of-inertia
    :accessor moments-of-inertia)
   (mass
    :initform 100
    :initarg :mass
    :accessor mass)
   (node-list    ;(x y z 1) in body coord for each node. Starts with (0 0 0 1).
    :initform '((0 0 0 1) (-0.5 -0.5 0.5 1) (0.5 -0.5 0.5 1)(0.5 -0.5 -0.5 1)
                          (-0.5 -0.5 -0.5 1)(-0.5  0.5 0.5 1) (0.5  0.5 0.5 1)
                          (0.5  0.5 -0.5 1)(-0.5  0.5 -0.5 1)(0 0 0 1)(0 0 3 1))
    :initarg :node-list
    :accessor node-list)
   (polygon-list
    :initform '((1 5 8 4)(5 6 7 8)(6 2 3 7)(2 1 4 3)(9 10))
    :initarg :polygon-list
    :accessor polygon-list)
   (transformed-node-list  ;(x y z 1) in earth coord for each node in node-list.
    :initform '((0 0 0 1) (-2 -2 0 1) (2 -2 0 1)(2 -2 -30 1)(-2 -2 -30 1)
                           (-2  2 0 1) (2  2 0 1)(2  2 -30 1)(-2  2 -30 1))
    :accessor transformed-node-list)
   (H-matrix
```

```
    :initform (unit-matrix 4)
    :accessor H-matrix)
   (time-stamp
    :accessor time-stamp)))

(defmethod initialize ((body rigid-body))
  (setf (transformed-node-list body) (node-list body))
  (setf (velocity-growth-rate body) (update-velocity-growth-rate body))
  (setf (posture-rate body) (earth-velocity body))
  (setf (time-stamp body) (get-internal-real-time))))

(defmethod set-transformed-node-list-z((body rigid-body) z)
   (setf (third(fourth (transformed-node-list body)))(- z))
   (setf (third(fifth (transformed-node-list body))) (- z))
   (setf (third(eighth (transformed-node-list body))) (- z))
   (setf (third(ninth (transformed-node-list body))) (- z))
   (transformed-node-list body)))

(defun transform (obj)
   (list 0 (first obj)(second obj)(third obj)))

(defun reverse-transform (obj)
   (list (second obj)(third obj)(fourth obj) 1))

(defmethod move ((body rigid-body) azimuth elevation roll x y z)
  (setf (posture body) (list x y z roll elevation azimuth))
  (setf (H-matrix body)
    (homogeneous-transform azimuth elevation roll x y z))
  (transform-node-list body))

(defmethod get-delta-t ((body rigid-body)) 0.005)

(defmethod forces-earth-to-body (Fx-e Fy-e Fz-e theta)
  (let* ((forces-earth (list Fx-e Fy-e Fz-e))

         (R-matrix (rotation-matrix 0 (- theta) 0))
         (forces-body (post-multiply R-matrix forces-earth)))

        forces-body))

(defmethod update-rigid-body ((body rigid-body))     ;Euler integration.
  (let* ((delta-t (get-delta-t body)))
       (update-posture body delta-t); EULER
       (update-velocity body delta-t); EULER
       (setf (H-matrix body) (homogeneous-transform (sixth (posture body))
          (fifth (posture body)) (fourth (posture body)) (first (posture body))
          (second (posture body)) (third (posture body))))
       (transform-node-list body)
       (update-velocity-growth-rate body)))

(defmethod update-velocity-growth-rate ((body rigid-body))
  (setf (velocity-growth-rate body) ;Assumes principal axis coordinates with
    (multiple-value-bind           ;origin at center of gravity of body.
      (Fx Fy Fz L M N   u v w p q r   Ix Iy Iz) ;Declares local variables.
      (values-list                              ;Values assigned.
```

```
        (append
          (forces-and-torques body) (velocity body) (moments-of-inertia body)))
      (list (+ (* v r) (* -1 w q) (/ Fx (mass body))
               (* *gravity* (first (third (H-matrix body)))))
            (+ (* w p) (* -1 u r) (/ Fy (mass body))
               (* *gravity* (second (third (H-matrix body)))))
            (+ (* u q) (* -1 v p) (/ Fz (mass body))
               (* *gravity* (third (third (H-matrix body)))))
            (/ (+ (* (- Iy Iz) q r) L) Ix)
            (/ (+ (* (- Iz Ix) r p) M) Iy)
            (/ (+ (* (- Ix Iy) p q) N) Iz)))))  ;Value returned.

(defmethod update-velocity ((body rigid-body) delta-t) ;Euler integration.
  (setf (velocity body)
    (vector-add (velocity body)
                (scalar-multiply delta-t (velocity-growth-rate body)))))

(defmethod update-posture ((body rigid-body) delta-t) ;Euler integration.
  (setf (posture-rate body) (earth-velocity body))
  (setf (posture body)
     (vector-add (posture body) (scalar-multiply delta-t (posture-rate body)))))

(defmethod transform-node-list ((body rigid-body))
  (setf (transformed-node-list body)
    (mapcar #'(lambda (node-location)
                (post-multiply (H-matrix body) node-location))
            (node-list body))))

(defconstant *gravity* 32.2185)

(defmethod earth-velocity ((body rigid-body))
  (let* ((linear-velocity (firstn 3 (velocity body)))
         (rotational-velocity (cdddr (velocity body)))
         (posture (posture body))
         (R-matrix (rotation-matrix (sixth posture) (fifth posture)
                                    (fourth posture)))
         (linear-earth-velocity (post-multiply R-matrix linear-velocity))
         (T-matrix (body-rate-to-euler-rate-matrix (sixth posture)
                     (fifth posture) (fourth posture)))
         (rotational-earth-velocity (post-multiply T-matrix
                                                   rotational-velocity)))
        (append linear-earth-velocity rotational-earth-velocity)))

(defun test-rigid-body ()
  (setf airplane-1 (make-instance 'rigid-body))
  (initialize airplane-1)
  (setf camera-1 (make-instance 'strobe-camera))
  (move camera-1 (- (/ pi 2)) 0  0 0 30 0 )
  (take-picture camera-1 airplane-1)
  (dotimes (i 20 'done) (update-rigid-body airplane-1))
  (take-picture camera-1 airplane-1))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;              - two Link Inverted Pendulum
;              - Massless leg and a single rigid body (fixed leg length)
;              - state vector x = (theta-1 theta-1-dot theta-2 theta-2-dot)
;              - state vector u = (M-1 M-2)
;                 - control torque at the ankle (M-1)
;                 - control torque at the hip (M-2)
;              - Newton-Euler Method
;                 -  3x3 matrix inversion to calculate forces and moments
;              - Heun Integration
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   (setf  l       0.5)
   (setf  r       3)
   (setf time-step 0.01)

(defun solve-three-equation-system (state-vector K-theta-1 K-theta-1-dot
                                                 K-theta-2 K-theta-2-dot)
   (setf  m       100)
   (setf  l       0.5)
   (setf  r       3)
   (setf  I       100)
   (setf  M-1 (compute-M-1 state-vector K-theta-1 K-theta-1-dot
                                        K-theta-2 K-theta-2-dot))
   (setf  M-2 (compute-M-2 state-vector K-theta-1 K-theta-1-dot
                                        K-theta-2 K-theta-2-dot))
   (setf  theta-1       .(first  state-vector))
   (setf  theta-1-dot    (second state-vector))
   (setf  theta-2        (third  state-vector))
   (setf  theta-2-dot    (fourth state-vector))
   (setf  g    32.2)

   (post-multiply
      (matrix-inverse
          (list (list  0     I     (* -1  l (sin (- theta-2 theta-1))))
                (list (* m  r (cos theta-1))  (* m  l (cos theta-2))
                      (* -1 (sin theta-1)))
                (list (* m  r (sin theta-1))  (* m  l (sin theta-2))
                      (cos theta-1))))

       (list  (/ (* (- (* M-2 (+ l r)) (* l M-1))(cos (- theta-2 theta-1))) r)
              (+ (* m r theta-1-dot theta-1-dot (sin theta-1))
                 (* m l theta-2-dot theta-2-dot (sin theta-2))
                 (/ (* (- M-1 M-2) (cos theta-1)) r))
              (+ (* -1 m r theta-1-dot theta-1-dot (cos theta-1))
                 (* -1 m l theta-2-dot theta-2-dot (cos theta-2))
                 (* m g)
                 (/ (* (- M-1 M-2) (sin theta-1)) r)))))


(defun euler-step-constant-leg (state-vector threeXthree-matrix)
  (mapcar '+ state-vector (scalar-multiply time-step
                            (derivative-state-vector
                             state-vector threeXthree-matrix))))
```

```
(defun heun-step-constant-leg (x K-theta-1 K-theta-1-dot
                                 K-theta-2 K-theta-2-dot)
  (setf threeXthree-matrix (solve-three-equation-system
                             x K-theta-1 K-theta-1-dot
                               K-theta-2 K-theta-2-dot))
  (mapcar '+ x (scalar-multiply (* time-step .5)
    (vector-add (derivative-state-vector x threeXthree-matrix)
                (derivative-state-vector
                     (euler-step-constant-leg x threeXthree-matrix)
                     threeXthree-matrix)))))

(defun derivative-state-vector (state-vector threeXthree-matrix)
           (list (second state-vector)
                 (first threeXthree-matrix)
                 (fourth state-vector)
                 (second threeXthree-matrix)))


(defun compute-M-1 (state-vector K-theta-1 K-theta-1-dot
                                 K-theta-2 K-theta-2-dot)
  (setf theta-1 (first state-vector))
  (setf theta-1-dot (second state-vector))
   (+ (* -1 K-theta-1 theta-1)(* -1 K-theta-1-dot theta-1-dot)))

(defun compute-M-2 (state-vector K-theta-1 K-theta-1-dot
                                 K-theta-2 K-theta-2-dot)
  (setf theta-2 (third state-vector))
  (setf theta-2-dot (fourth state-vector))
   (+ (* -1 K-theta-2 theta-2)(* -1 K-theta-2-dot theta-2-dot)))

(defun compute-y (state-vector)
   (setf  theta-1       (first  state-vector))
   (setf  theta-1-dot   (second state-vector))
   (setf  theta-2       (third  state-vector))
   (setf  theta-2-dot   (fourth state-vector))
   (setf  l       0.5)
   (setf  r       3)
   (+ (* r (sin theta-1))
      (* l (sin theta-2))))

(defun compute-z (state-vector)
   (setf  theta-1       (first  state-vector))
   (setf  theta-1-dot   (second state-vector))
   (setf  theta-2       (third  state-vector))
   (setf  theta-2-dot   (fourth state-vector))
   (setf  l       0.5)
   (setf  r       3)
   (+ (* r (cos theta-1))
      (* l (cos theta-2))))


(defun draw-inverted-pendulum (halt-time K-theta-1 K-theta-1-dot
                                         K-theta-2 K-theta-2-dot)
    (setf robot-body (make-instance 'rigid-body))
```

```
    (initialize  robot-body)

    (setf robot-leg (make-instance 'rigid-body))
    (initialize  robot-leg)


    (setf camera-1 (make-instance 'strobe-camera))
    (move camera-1 (deg-to-rad 0)(deg-to-rad 0)(deg-to-rad 0) -10 0 0)

    (setf graph-1-window (make-window-graph-1))
    (draw-coordinate-axes graph-1-window)

    (setf graph-2-window (make-window-graph-2))
    (draw-coordinate-axes graph-2-window)

    (do ((x '(0.5 0 1.0 0) (heun-step-constant-leg x K-theta-1 K-theta-1-dot
                                                   K-theta-2 K-theta-2-dot))
        (old-x '(0.5 0 1.0 0) x)
        (time 0 (+ time 1)))
      ((> time halt-time) 'done)

      (move  robot-body (deg-to-rad 0)(deg-to-rad 0) (third x)
                        0            (compute-y x) (* -1(compute-z x)))

      (setf (transformed-node-list robot-leg) (list (list 0 0 0 1)
                                              (list 0 (* r (sin (first x)))
                                              (* -1  r (cos (first x))) 1)
                                              (list 0 0 0 1)))
      (setf (polygon-list robot-leg) (list (list 2 1)))

      (cw:clear (camera-window camera-1))

      (take-picture camera-1 robot-body)
      (take-picture camera-1 robot-leg)
      (draw-line-in-window graph-1-window 80 (list (* time-step (- time 1))
                                                   (first old-x))
                                             (list (* time-step time)
                                                   (first x)))
      (draw-line-in-window graph-2-window 80 (list (* time-step (- time 1))
                                                   (third old-x))
                                             (list (* time-step time)
                                                   (third x)))))

(defun make-window-graph-1 ()
  (cw:make-window-stream :borders 5
                         :left 0
                         :bottom 50
                         :width 450
                         :height 450
                         :title "Leg Angle   (theta-1) NEWTON-EULER"
                         :activate-p t))  ;Make window visible.

(defun make-window-graph-2 ()
  (cw:make-window-stream :borders 5
                         :left 0
```

```
:bottom 550
:width 450
:height 450
:title "Body Attitude   (theta-2) NEWTON-EULER"
:activate-p t))  ;Make window visible.
```

```
;; ***************************************************************************
;; File         : load-euler-files.cl
;; Author       : Mehmet Bediz
;;              : Naval Postgraduate School
;;              : Monterey, CA 93943
;; Summary      : This file contains functions to load files.
;; ***************************************************************************

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;        Load files
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(load "harmonic-equation.cl")
(load "robot-kinematics.cl")
(load "euler-angle-rigid-body.cl")
(load "strobe-camera.cl")
```

```
;; ***************************************************************************
;; File        : strobe-camera.cl
;; Author      : Dr. R. McGhee
;;             : Naval Postgraduate School
;;             : Monterey, CA 93943
;; Summary     : This file contains strobe-camera class definitions.
;; ***************************************************************************
(require :xcw)
(cw:initialize-common-windows)

(defclass strobe-camera (rigid-body)
  ((focal-length
    :accessor focal-length
    :initform 6)
   (camera-window
    :accessor camera-window
    :initform (cw:make-window-stream    :borders 5
                                        :left 470
                                        :bottom 500
                                        :width 650
                                        :height 650
                                        :title "Two Link Inverted Pendulum
                                               (Constant Length Massless Leg)
                                                NEWTON-EULER"
                                        :activate-p t))
   (H-matrix
    :initform (homogeneous-transform .3 -.3 0 -300 -90 -90))
   (inverse-H-matrix
    :accessor inverse-H-matrix
    :initform (inverse-H (homogeneous-transform .3 -.3 0 -300 -90 -90)))
   (enlargement-factor
    :accessor enlargement-factor
    :initform 30)))

(defmethod move ((camera strobe-camera) azimuth elevation roll x y z)
  (setf (H-matrix camera) (homogeneous-transform azimuth elevation roll x y z))
  (setf (inverse-H-matrix camera) (inverse-H (H-matrix camera))))

(defmethod take-picture ((camera strobe-camera) (body rigid-body))
  (let ((camera-space-node-list (mapcar #'(lambda (node-location)
         (post-multiply (inverse-H-matrix camera) node-location))
                        (transformed-node-list body))))
    (dolist (polygon (polygon-list body))
      (clip-and-draw-polygon camera polygon camera-space-node-list))))

(defmethod clip-and-draw-polygon
  ((camera strobe-camera) polygon node-coord-list)
  (do* ((initial-point (nth (first polygon) node-coord-list))
        (from-point initial-point to-point)
        (remaining-nodes (rest polygon) (rest remaining-nodes))
        (to-point (nth (first remaining-nodes) node-coord-list)
                  (if (not (null (first remaining-nodes)))
                      (nth (first remaining-nodes) node-coord-list))))
       ((null to-point)
        -      (draw-clipped-projection camera from-point initial-point))
```

```
      (draw-clipped-projection camera from-point to-point)))

(defmethod draw-clipped-projection ((camera strobe-camera) from-point to-point)
  (cond ((and (<= (first from-point) (focal-length camera))
              (<= (first to-point) (focal-length camera))) nil)
        ((<= (first from-point) (focal-length camera))
         (draw-line-in-camera-window camera
           (perspective-transform camera (from-clip camera from-point to-point))
           (perspective-transform camera to-point)))
        ((<= (first to-point) (focal-length camera))
         (draw-line-in-camera-window camera
           (perspective-transform camera from-point)
           (perspective-transform camera (to-clip camera from-point to-point))))
        (t (draw-line-in-camera-window camera
           (perspective-transform camera from-point)
           (perspective-transform camera to-point)))))

(defmethod from-clip ((camera strobe-camera) from-point to-point)
  (let ((scale-factor (/ (- (focal-length camera) (first from-point))
                         (- (first to-point) (first from-point)))))
       (list (+ (first from-point)
                (* scale-factor (- (first to-point) (first from-point))))
             (+ (second from-point)
                (* scale-factor (- (second to-point) (second from-point))))
             (+ (third from-point)
                (* scale-factor (- (third to-point) (third from-point)))) 1)))

(defmethod to-clip ((camera strobe-camera) from-point to-point)
  (from-clip camera to-point from-point))

(defmethod draw-line-in-camera-window ((camera strobe-camera) start end)
  (cw:draw-line (camera-window camera)
                (cw:make-position :x (+ 150 (first start))
                                  :y (+ 150 (second start)))
                (cw:make-position :x (+ 150 (first end))
                                  :y (+ 150 (second end)))
                :brush-width 0))

(defmethod perspective-transform ((camera strobe-camera) point-in-camera-space)
  (let* ((enlargement-factor (enlargement-factor camera))
              (focal-length (focal-length camera))
              (x (first point-in-camera-space))  ;x axis is along optical axis
              (y (second point-in-camera-space)) ;y is out right side of camera
              (z (third point-in-camera-space))) ;z is out bottom of camera
             (list (+ (round (* enlargement-factor (/ (* focal-length y) x)))
                      150)                      ;to right in camera window
              (+ 150 (round (* enlargement-factor (/ (* focal-length (- z)) x))
                        )))))  ;up in camera window

(defun test-camera (z theta) ;Produces top view of default rigid-body.
  (setf robot-leg (make-instance 'rigid-body))
  (initialize  robot-leg)
  (set-transformed-node-list-z robot-leg z)
  (set-transformed-node-list-theta robot-leg theta)
  (setf camera-1 (make-instance 'strobe-camera))
```

```
  (move camera-1 (deg-to-rad 0)(deg-to-rad 0)(deg-to-rad 0) -30 0 0)
  (take-picture camera-1 robot-leg))

(defun deg-to-rad (angle) (* .01745329251994329 angle))

(defun animation ()

    (do ((theta-loop 0 (+ theta-loop 1)))
    ((> theta-loop 90) 'end)
   (test-camera 30 theta-loop)))
```

APPENDIX B: KINEMATIC SIMULATION SOFTWARE

```
// ************************************************************************
// File           : main.C
// Author         : Mehmet Bediz
//                : Naval Postgraduate School
//                : Monterey, CA 93943
// Created        : November 1996
// Summary        : This file contains the main function and three motion
//                : functions of the kinematic model Dynaman: forward stepping,
//                : upward stepping, and jumping.
// ************************************************************************

#include <string.h>
#include "main.h"
void step_forward(int number_of_steps, float X, float Y, float Z);
void step_upward(int number_of_steps, float X, float Y, float Z);
void jump(float X,float Y,float Z);

// Speed of the animation can be controlled by changing delta_t
float delta_t = 2 * 3.0;

   pfNode          *root;
   pfDCS           *dcs;
   pfMatrix        mat, orbit;
   pfSphere        sphere;
   pfNode          *model1, *model2, *model3, *model4;
   pfNode          *model8, *model10;
   pfNode          *model11, *model12, *model13, *model14;
   pfDCS           *node11, *node12, *node13, *node14;
   pfDCS           *node1, *node2, *node3, *node4;
   pfDCS     *node8, *node10;
   pfDCS           *dcs0;
   pfDCS           *dcs1, *dcs2, *dcs3;
   pfDCS           *dcs4, *dcs5, *dcs6;
   pfDCS     *dcs7, *dcs8, *dcs9, *dcs10;
   pfDCS     *dcs11, *dcs12, *dcs13;
   pfDCS     *dcs14, *dcs15, *dcs16;

   pfDCS     *dcs30, *dcs31, *dcs32, *dcs33;
   pfDCS     *dcs34, *dcs35, *dcs36;
   pfDCS     *dcs37, *dcs38, *dcs39;

   char            *file1, *file2, *file3, *file4;
   char            *file8, *file10;
   char            *file11, *file12, *file13, *file14;

   pfLightSource   *light;
   pfMatrix        lightMat;
   float           d;
   pfTexture       *tex;
   pfFrustum       *Frust;
   void            *arena;
   pfDCS           *lightDCS;
```

```
   int         i;
   float       final_position;

//------------------------------------------------------------------------
//  Function:       main
//  Returns:        None
//  Parameters:     None
//  Summary:        Initializes IRIS Performer™, loads body parts from poly format
//                  files to DCS nodes, creates the channels at the upper right
//                  corner of the window, calls step_forward, step_upward, and
//                  jump functions with the number of steps and the initial
//                  positions.
//------------------------------------------------------------------------

void
main (int argc, char *argv[])
{

   /* choose default objects of none specified */
   file1 =  "lowerleg.poly";
   file2 =  "upperleg.poly";
   file3 =  "head.iv";
   file4 =  "torso.poly";
   file8 =  "foot.poly";
   file10 = "floor.iv";
   file11 = "rightupperarm.poly";
   file12 = "lowerarm.poly";
   file13 = "hand.poly";
   file14 = "leftupperarm.poly";

   if ( ! strcmp(argv[1],"slow")){
     delta_t = 2 * 0.5;
   }

   #ifndef IRISGL
     printf("Sorry, shadows doesn't work in OPENGL\n");
     return 0;
   #endif

   /* Initialize Performer */
   pfInit();

   /* Use default multiprocessing mode based on number of
    * processors.
    */
   pfMultiprocess(PFMP_DEFAULT);

   /** allocate shared memory   **/
   InitShared();

   /* Configure multiprocessing mode and start parallel
    * processes.
    */
   pfConfig();
```

```
/** Initialize Performer utility and GUI functions **/
pfuInitUtil();

pfInitClock(0.0f);
Shared->simTime = pfGetTime();
srand(Shared->simTime*10000000);

/* Append to PFPATH additional standard directories where
 * geometry and textures exist
 */
pfFilePath(".:/usr/share/Performer/data:/tmp_mnt/workb/bediz/Performer/data");

/* Do not use FLAT_ primitives because they look bad
 * with local lighting.
*/
pfdBldrMode(PFDBLDR_MESH_LOCAL_LIGHTING, 1);

/* Read a single file, of any known type. */
if ((root = pfdLoadFile("text.iv")) == NULL)
{
                pfExit();
                exit(-1);
}

/* Load the files */
if ((model1 = pfdLoadFile(file1)) == NULL)
{
                pfExit();
                exit(-1);
}
if ((model2 = pfdLoadFile(file2)) == NULL)
{
                pfExit();
                exit(-1);
}
if ((model3 = pfdLoadFile(file3)) == NULL)
{
                pfExit();
                exit(-1);
}
if ((model4 = pfdLoadFile(file4)) == NULL)
{
                pfExit();
                exit(-1);
}
if ((model8 = pfdLoadFile(file8)) == NULL)
{
                pfExit();
                exit(-1);
}
if ((model10 = pfdLoadFile(file10)) == NULL)
{
                pfExit();
                exit(-1);
}
```

```
if ((model11 = pfdLoadFile(file11)) == NULL)
{
                pfExit();
                exit(-1);
}
if ((model12 = pfdLoadFile(file12)) == NULL)
{
                pfExit();
                exit(-1);
}
if ((model13 = pfdLoadFile(file13)) == NULL)
{
                pfExit();
                exit(-1);
}
if ((model14 = pfdLoadFile(file14)) == NULL)
{
                pfExit();
                exit(-1);
}

/* Create and attach loaded subgraph to a pfScene. */
Shared->scene = new pfScene;
dcs = new pfDCS;
dcs->addChild(root);

Shared->scene->addChild(dcs);
/* determine extent of scene's geometry */
Shared->scene->getBound(&(Shared->bsphere));

/****************** MAIN WINDOW ***************/
/* Configure and open GL window */
Shared->p = pfGetPipe(0);
Shared->p2 = pfGetPipe(0);
Shared->pw = new pfPipeWindow(Shared->p);
Shared->pw->setName("DYNAMAN");
Shared->pw->setConfigFunc(OpenPipeWin);
Shared->pw->setOriginSize(120, 120, 1100, 1100);
Shared->pw->config();

/*   initializes mouse and keyboard inputs to be read from window  */
pfuInitInput(Shared->pw, PFUINPUT_GL);
pfuInitGUI(Shared->pw);
pfuEnableGUI(TRUE);

/* Create and configure a pfChannel. */
Shared->chan =  new pfChannel(Shared->p);
Shared->chan->setScene(Shared->scene);
Shared->chan->setNearFar(1.0f, 5.0f * Shared->bsphere.radius);
Shared->chan->setFOV(45.0f, 0.0f);
Shared->view.xyz.set(1.3f * Shared->bsphere.radius,
                -2.5f * Shared->bsphere.radius,
                 3.0f * Shared->bsphere.radius);
Shared->view.hpr.set(0.0f, -45.0f, 0.0f);
Shared->chan->setView(Shared->view.xyz, Shared->view.hpr);
```

```
Shared->chan->setTravFunc(PFTRAV_DRAW,DrawChannel);

/******** FIRST CHANNEL ************************/
Shared->chan2[0] = new pfChannel(Shared->p);
Shared->pw->addChan(Shared->chan2[0]);
Shared->chan2[0]->setTravFunc(PFTRAV_DRAW,DrawChannel);
Shared->chan2[0]->setScene(Shared->scene);
Shared->chan2[0]->setNearFar(1.0f, 5.0f * Shared->bsphere.radius);
Shared->chan2[0]->setFOV(45.0f, 0.0f);
Shared->view.xyz.set(       Shared->bsphere.radius,
                -4.0f * Shared->bsphere.radius,
                 0.8f * Shared->bsphere.radius);
Shared->view.hpr.set(0.0f, 0.0f, 0.0f);
Shared->chan2[0]->setView(Shared->view.xyz, Shared->view.hpr);

/********   SECOND CHANNEL   ************************/

Shared->chan2[1] = new pfChannel(Shared->p);
Shared->pw->addChan(Shared->chan2[1]);
Shared->chan2[1]->setTravFunc(PFTRAV_DRAW,DrawChannel);
Shared->chan2[1]->setScene(Shared->scene);
Shared->chan2[1]->setNearFar(1.0f, 5.0f * Shared->bsphere.radius);
Shared->chan2[1]->setFOV(45.0f, 0.0f);
Shared->view.xyz.set(1.3  * Shared->bsphere.radius,
               -2.5f * Shared->bsphere.radius,
                3.0f * Shared->bsphere.radius);
Shared->view.hpr.set(0.0f, -45.0f, 0.0f);
Shared->chan2[1]->setView(Shared->view.xyz, Shared->view.hpr);

/********   THIRD CHANNEL    ************************/

Shared->chan2[2] = new pfChannel(Shared->p);
Shared->pw->addChan(Shared->chan2[2]);
Shared->chan2[2]->setTravFunc(PFTRAV_DRAW,DrawChannel);
Shared->chan2[2]->setScene(Shared->scene);
Shared->chan2[2]->setNearFar(1.0f, 5.0f * Shared->bsphere.radius);
Shared->chan2[2]->setFOV(45.0f, 0.0f);
Shared->view.xyz.set(  1.6   * Shared->bsphere.radius,
               -0.15f * Shared->bsphere.radius,
                2.5f  * Shared->bsphere.radius);
Shared->view.hpr.set(0.0f, -90.0f, 0.0f);
Shared->chan2[2]->setView(Shared->view.xyz,Shared-> view.hpr);

/********  FOURTH CHANNEL    ************************/

Shared->chan2[3] = new pfChannel(Shared->p);
Shared->pw->addChan(Shared->chan2[3]);
Shared->chan2[3]->setTravFunc(PFTRAV_DRAW,DrawChannel);
Shared->chan2[3]->setScene(Shared->scene);
Shared->chan2[3]->setNearFar(1.0f, 5.0f * Shared->bsphere.radius);
Shared->chan2[3]->setFOV(45.0f, 0.0f);
Shared->view.xyz.set(  4.0f * Shared->bsphere.radius,
               -0.5f * Shared->bsphere.radius,
                0.5 * Shared->bsphere.radius);
Shared->view.hpr.set(90.0f, 0.0f, 0.0f);
```

```
Shared->chan2[3]->setView(Shared->view.xyz, Shared->view.hpr);
Shared->chan2[3]->setViewport(0.7, 0.85, 0.7, 0.85);
Shared->chan2[2]->setViewport(0.85, 1.0, 0.7, 0.85);
Shared->chan2[1]->setViewport(0.85, 1.0, 0.85, 1.0);
Shared->chan2[0]->setViewport(0.7, 0.85, 0.85, 1.0);

orbit.makeRot(1.0f, 0.0f, 0.0f, 1.0f);

/* scale models to unit size */
node1 = new pfDCS;
node1->addChild(model1);
model1->getBound(&sphere);
if (sphere.radius > 0.0f)
                    node1->setScale(1.0f/sphere.radius);

node2 = new pfDCS;
node2->addChild(model2);
model2->getBound(&sphere);
if (sphere.radius > 0.0f)
                     node2->setScale(1.0f/sphere.radius);

node3 = new pfDCS;
node3->addChild(model3);
model2->getBound(&sphere);
if (sphere.radius > 0.0f)
                     node3->setScale(1.0f/sphere.radius);

node4 = new pfDCS;
node4->addChild(model4);
model4->getBound(&sphere);
if (sphere.radius > 0.0f)
                     node4->setScale(1.0f/sphere.radius);

node8 = new pfDCS;
node8->addChild(model8);
model8->getBound(&sphere);
if (sphere.radius > 0.0f)
                     node8->setScale(1.0f/sphere.radius);

node10 = new pfDCS;
node10->addChild(model10);
model10->getBound(&sphere);
if (sphere.radius > 0.0f)
                     node10->setScale(1.0f/sphere.radius);

node11 = new pfDCS;
node11->addChild(model11);
model11->getBound(&sphere);
if (sphere.radius > 0.0f)
                     node10->setScale(1.0f/sphere.radius);

node12 = new pfDCS;
node12->addChild(model12);
model12->getBound(&sphere);
if (sphere.radius > 0.0f)
```

```
                    node10->setScale(1.0f/sphere.radius);

node13 = new pfDCS;
node13->addChild(model13);
model13->getBound(&sphere);
if (sphere.radius > 0.0f)
                    node10->setScale(1.0f/sphere.radius);

node14 = new pfDCS;
node14->addChild(model14);
model14->getBound(&sphere);
if (sphere.radius > 0.0f)
                    node10->setScale(1.0f/sphere.radius);

/* Lighting without shadowing   */
// put a default light source in the scene
Shared->scene->addChild(new pfLightSource);

arena = pfGetSharedArena();

// Create and configure shadow frustum. Fit frustum tightly to scene.
Frust = new(arena) pfFrustum;
Frust->makeSimple(FOV);

d = Shared->bsphere.radius / sinf(PF_DEG2RAD(FOV/2.0f));
d = PF_MAX2(d, NEAR + Shared->bsphere.radius);

Frust->setNearFar(NEAR, d + 1.1f * Shared->bsphere.radius);
tex = initSpotTex();

light = new pfLightSource;
light->setMode(PFLS_PROJTEX_ENABLE, 1);
light->setAttr(PFLS_PROJ_FRUST, Frust);
light->setAttr(PFLS_PROJ_TEX, tex);
light->setColor(PFLT_DIFFUSE, 1.0f, 1.0f, 1.0f);
light->setVal(PFLS_INTENSITY, 3.0f);
light->setPos(17.0  , -7.0 , 13.9, 1.0f);// Make light local
light->setSpotDir(0.0f, 0.0f, -10.0f);

// Make DCS to move lights around
lightDCS = new pfDCS;
lightDCS->addChild(light);

/*  Main DCS   */
dcs0 = new pfDCS;

/* Left Foot   */
dcs1 = new pfDCS;
dcs1 -> addChild(node1);

/* Left lowerleg */

/* Right lowerleg */
dcs4 = new pfDCS;
dcs4 -> addChild(node1);
```

```
/* Left upperleg  */
dcs2 = new pfDCS;
dcs2 -> addChild(node2);

/* Right upperleg */
dcs5 = new pfDCS;
dcs5 -> addChild(node2);

/* Floor */
dcs10 = new pfDCS;
dcs10 -> addChild(node10);
dcs30 = new pfDCS;
dcs30 -> addChild(node10);
dcs31 = new pfDCS;
dcs31 -> addChild(node10);
dcs32 = new pfDCS;
dcs32 -> addChild(node10);
dcs33 = new pfDCS;
dcs33 -> addChild(node10);
dcs34 = new pfDCS;
dcs34 -> addChild(node10);
dcs35 = new pfDCS;
dcs35 -> addChild(node10);
dcs36 = new pfDCS;
dcs36 -> addChild(node10);
dcs37 = new pfDCS;
dcs37 -> addChild(node10);
dcs38 = new pfDCS;
dcs38 -> addChild(node10);
dcs39 = new pfDCS;
dcs39 -> addChild(node10);

/*  Torso (Two parts)  */
dcs6 = new pfDCS;
dcs6 -> addChild(node4);
dcs0 -> addChild(dcs6);

dcs3 = new pfDCS;
dcs3 -> addChild(node4);
dcs0 -> addChild(dcs3);

/*  Left Legs  */
dcs2 -> addChild(dcs1);
dcs0 -> addChild(dcs2);

/*  Right  Legs  */
dcs5 -> addChild(dcs4);
dcs0 -> addChild(dcs5);

/*  Feet */
dcs8 = new pfDCS;
dcs8 -> addChild(node8);
dcs9 = new pfDCS;
dcs9 -> addChild(node8);
```

```
dcs1 -> addChild(dcs8);
dcs4 -> addChild(dcs9);

/*  Head  */
dcs7 = new pfDCS;
dcs7 -> addChild(node3);
dcs3 -> addChild(dcs7);

/*  Right upper arm  */
dcs11 = new pfDCS;
dcs11 -> addChild(node11);
dcs0 -> addChild(dcs11);

/*  Right lower arm  */
dcs12 = new pfDCS;
dcs12 -> addChild(node12);
dcs11 -> addChild(dcs12);

/*  Right hand  */
dcs13 = new pfDCS;
dcs13 -> addChild(node13);
dcs12 -> addChild(dcs13);

/*  Left upper arm  */
dcs14 = new pfDCS;
dcs14 -> addChild(node14);
dcs0 -> addChild(dcs14);

/*  Left lower arm  */
dcs15 = new pfDCS;
dcs15 -> addChild(node12);
dcs14 -> addChild(dcs15);

/*  Left hand  */
dcs16 = new pfDCS;
dcs16 -> addChild(node13);
dcs15 -> addChild(dcs16);

//  Stairs
dcs30 -> addChild(dcs31);
dcs31 -> addChild(dcs32);
dcs32 -> addChild(dcs33);
dcs33 -> addChild(dcs34);
dcs34 -> addChild(dcs35);
dcs35 -> addChild(dcs36);
dcs36 -> addChild(dcs37);
dcs37 -> addChild(dcs38);
dcs38 -> addChild(dcs39);

Shared->scene-> addChild(dcs0);
Shared->scene-> addChild(dcs30);

/* Create "floor letters"  */
dcs -> addChild(initFloor(&Shared->bsphere));
```

```
/*  Floor       */
dcs10->setScale(2.0f);
dcs10->setRot(0.0, -90.0 ,0.0);
dcs10->setTrans(-1.0  , -11.0 , 0.05);

//  Draw steps
dcs30->setScale(0.5f);
dcs30->setRot(-90.0, -90.0 ,0.0);
dcs30->setTrans(13.0  , 0.0 , 0.05);
dcs31->setTrans(0.0  , -2.0 , 5.4);
dcs32->setTrans(0.0  , -2.0 , 4.1);
dcs33->setTrans(0.0  , -1.9 , 4.1);
dcs34->setTrans(0.0  , -1.9 , 4.1);
dcs35->setTrans(0.0  , -1.9 , 4.1);
dcs36->setTrans(0.0  , -1.8 , 4.1);
dcs37->setTrans(0.0  , -1.8 , 4.1);
dcs38->setTrans(0.0  , -1.8 , 4.1);
dcs39->setTrans(0.0  , -1.7 , 4.1);

/*     Set up initial/default view  */
MakeGUI();
int temp =  system("sfplay runaway.wav &");
if (-1 == temp){
  printf("Can't play \n");
}
else{
 printf("Playing %i \n",temp);
}
for (int forever = 0; forever < 50; forever ++){
  step_forward(6, -9.8  , -7.0 , 3.9);
  step_upward(5, 13.3 , -7.0 , 3.9);
  jump(31.7, -7.0,12.496556);
  step_forward(4, 39.137990  , -7.0 , 4.670563);
}
// Kill music
system("music_killer");

/*  Terminate parallel processes and exit. */
pfExit();
}
```

```
//---------------------------------------------------------------------
// Function:      step_forward
// Returns:       None
// Parameters:    Number of steps needs to be taken, initial position
// Summary:       Computes the joint angles according to the position
//                of the end effector(foot) by using the inverse kinematic
//                equations of the three link planar manipulator. The
//                algorithm for the path of the foot is described in
//                Chapter IV of this thesis as "Stepping Forward Algorithm".
//---------------------------------------------------------------------


void step_forward(int number_of_steps,float X,float Y,float Z){

  float l1 = 1.0f;
  float l2 = 1.0f;
  float c1_left,c1_right;
  float theta1,theta2;
  float k1_left,k2_left;
  float k1_right,k2_right;

  float x_left =  1.732f;
  float y_left =  0.0f;
  float x_right =  1.732f;
  float y_right =  0.0f;

//  INITIAL  HALF STEP
  for (float z = 32  ; z < 43 ; z += 0.5 * delta_t)
  {
    /*  Go to sleep until next frame time. */
    pfSync();
    Shared->simTime = pfGetTime();

    /*   Main Body   */
    dcs0->setRot(90.0, 90.0 , 0);
    dcs0->setTrans(X,Y,Z);

    /*   Head   */
    dcs7->setTrans(0.0  , 0.4 , 0.0);

    /*   Torso   */
    dcs6->setScale(2.0f);
    dcs6->setTrans(0.3  , 2.1 , -0.005);
    dcs3->setScale(2.0f);
    dcs3->setTrans(0.3  , 2.1 , 0.005);

     /* Left foot   */
    dcs9->setScale(0.7f);
    dcs9->setRot(0, 0 , 180.0);
    dcs9->setTrans(0  , -1.7, 0);

    /* Right foot   */
    dcs8->setScale(0.7f);
    dcs8->setRot(0, 0 , 180.0);
    dcs8->setTrans(0  , -1.7, 0);
```

```
    /* Left lowerleg   */
    dcs4->setTrans(0  , -1.7, 0);

    /* Left upperleg   */
    dcs5->setTrans(0.6,  0,   0);

    /* Right lowerleg   */
    dcs1->setTrans(0  , -1.7, 0);

    /* Right upper arm   */
    dcs11->setScale(0.25f);
    dcs11->setTrans(-1.0 , 1.7, 0);

    /* Right lower arm   */
    dcs12->setTrans(0.4 , -4.6, 0.0);

    /* Right hand   */
    dcs13->setTrans(0.0 , -4.2, 0.0);
    dcs13->setRot(0, 0 , 180.0);

    /* Left upper arm   */
    dcs14->setScale(0.25f);
    dcs14->setTrans(1.6  , 1.8, 0);

    /* Left lower arm   */
    dcs15->setRot(0, 0 , 180.0);
    dcs15->setTrans(-0.5 , -4.6, 0.0);

    /* Left hand   */
    dcs16->setTrans(-0.2 , -4.6, 0.0);

////////////////////////////////////////////////////////////
//              Inverse Kinematics
//
////////////////////////////////////////////////////////////
//  SECOND STEP
//  RIGHT LEG STEP

    x_right = 0.95f * cos((64 - 2* z) * DEG_TO_RAD )*
              (l1 + l2) * cos((64 - 2* z) * DEG_TO_RAD);
    y_right = 0.95f * cos((64 - 2* z) * DEG_TO_RAD )*
              (l1 + l2) * sin((64 - 2* z) * DEG_TO_RAD);

    c1_right= ((x_right*x_right) + (y_right*y_right)
        - (l1*l1) - (l2*l2))/(2 * l1 *l2);
    theta2 = (1 *acos(c1_right));

    k1_right = l1 + (l2 * cos(theta2));
    k2_right = l2 * sin(theta2);

    theta1 = atan(y_right/x_right) - atan(k2_right/k1_right);

    // Right Leg
    dcs2->setRot(0 ,(57.3 * theta1),0 );
    dcs1->setRot(0 ,(57.3 * theta2), 0);
```

```
    // Left Arm
    dcs14->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
    dcs15->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);

//   LEFT LEG STEP

    x_left = cos((-64 + 2* z) * DEG_TO_RAD ) * (l1 + l2) *
          cos((-64 + 2* z) * DEG_TO_RAD);
    y_left = cos((-64 + 2* z) * DEG_TO_RAD ) * (l1 + l2) *
          sin((-64 + 2* z) * DEG_TO_RAD);
    c1_left= ((x_left*x_left) + (y_left*y_left)
           - (l1*l1) - (l2*l2))/(2 * l1 *l2);
   theta2 = (1 *acos(c1_left));

   k1_left = l1 + (l2 * cos(theta2));
   k2_left = l2 * sin(theta2);
   theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);

   // Left leg
   dcs5->setRot(0 ,(57.3 * theta1), 0);
   dcs4->setRot(0 ,(57.3 * theta2), 0);

   // Right arm
   dcs11->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0);
   dcs12->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f),0);

   // Torso rotation
   if(z < 31){
     dcs3->setRot((-(z - 21)/3.0),0 ,0);
     dcs6->setRot((-(z  -21)/3.0),0 ,0);
   }
   else{
     dcs3->setRot(((-20 + ((z- 21)))/3.0),0 ,0);
     dcs6->setRot(((-20 + ((z- 21)))/3.0),0 ,0);
   }

   dcs0->setTrans(X + (z - 32) * 0.1  , Y ,
         (Z + 0.2) + 0.08f * cos(2.0f * 3.14159f * (z - 41) / 21.0f));

   UpdateView();
   UpdateGUI();
   pfFrame();
 }//End of second for

for( i = 1 ; i < number_of_steps ; i++){

   for ( z = (i-1)*42 + 1 ; z < (i-1)*42 + 22 ; z += 0.5 * delta_t){

       /*  Go to sleep until next frame time. */
       pfSync();
       Shared->simTime=pfGetTime();


       /*    Main Body    */
```

```
dcs0->setRot(90.0, 90.0 , 0);
dcs0->setTrans(X,Y,Z);

/*   Head   */
dcs7->setTrans(0.0  , 0.4 , 0.0);

/*   Torso   */
dcs6->setScale(2.0f);
dcs6->setTrans(0.3  , 2.1 , -0.005);
dcs3->setScale(2.0f);
dcs3->setTrans(0.3  , 2.1 , 0.005);

/* Left foot   */
dcs9->setScale(0.7f);
dcs9->setRot(0, 0 , 180.0);
dcs9->setTrans(0  , -1.7, 0);

/* Right foot   */
dcs8->setScale(0.7f);
dcs8->setRot(0, 0 , 180.0);
dcs8->setTrans(0  , -1.7, 0);

/* Left lowerleg   */
dcs4->setTrans(0  , -1.7, 0);

/* Left upperleg   */
dcs5->setTrans(0.6,  0,   0);

/* Right lowerleg   */
dcs1->setTrans(0  , -1.7, 0);

/* Right upper arm   */
dcs11->setScale(0.25f);
dcs11->setTrans(-1.0 , 1.7, 0);

/* Right lower arm   */
dcs12->setTrans(0.4 , -4.6, 0.0);

/* Right hand   */
dcs13->setTrans(0.0 , -4.2, 0.0);
dcs13->setRot(0, 0 , 180.0);

/* Left upper arm   */
dcs14->setScale(0.25f);
dcs14->setTrans(1.6  , 1.8, 0);

/* Left lower arm   */
dcs15->setRot(0, 0 , 180.0);
dcs15->setTrans(-0.5 , -4.6, 0.0);

/* Left hand   */
dcs16->setTrans(-0.2 , -4.6, 0.0);
```

```
//////////////////////////////////////////////////////
//              Inverse Kinematics
//////////////////////////////////////////////////////
//  FIRST STEP
//   LEFT LEG
      x_left = 0.95f * cos((22 - 2* (z- 42 *(i - 1))) * DEG_TO_RAD )
           *(l1 + l2) * cos((22 - 2* (z- 42 *(i - 1))) * DEG_TO_RAD);
      y_left = 0.95f * cos((22 - 2* (z- 42 *(i - 1))) * DEG_TO_RAD )
           *(l1 + l2) * sin((22 - 2* (z- 42 *(i - 1))) * DEG_TO_RAD);

      c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
            - (l2*l2))/(2 * l1 *l2);
      theta2 = (1 *acos(c1_left));
      k1_left = l1 + (l2 * cos(theta2));
      k2_left = l2 * sin(theta2);
      theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);

      dcs5->setRot(0 ,(57.3 * theta1),0 );
      dcs4->setRot(0 ,(57.3 * theta2) , 0);

      // Right Arm
      dcs11->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
      dcs12->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);

      // Right Leg
      x_right = cos((-22 + 2 * (z- 42 *(i - 1))) * DEG_TO_RAD )
        * (l1 + l2) * cos((-22 + 2 * (z- 42 *(i - 1))) * DEG_TO_RAD);
      y_right = cos((-22 + 2 * (z- 42 *(i - 1))) * DEG_TO_RAD )
        * (l1 + l2) * sin((-22 + 2 * (z- 42 *(i - 1))) * DEG_TO_RAD);
      c1_right= ((x_right*x_right) + (y_right*y_right)
        - (l1*l1) - (l2*l2))/(2 * l1 *l2);
      theta2 = (1 *acos(c1_right));

      k1_right = l1 + (l2 * cos(theta2));
      k2_right = l2 * sin(theta2);
      theta1 = atan(y_right/x_right) - atan(k2_right/k1_right);

      dcs2->setRot(0 ,(57.3 * theta1), 0);
      dcs1->setRot(0 ,(57.3 * theta2), 0);

      // Left Arm
      dcs14->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
      dcs15->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);

      // Torso rotation
      if((z- 42 *(i - 1)) < 11){
        dcs3->setRot(((z- 42 *(i - 1))/3.0),0 ,0);
        dcs6->setRot(((z- 42 *(i - 1))/3.0),0 ,0);
      }
      else{
        dcs3->setRot(((20 - (z- 42 *(i - 1)))/3.0),0 ,0);
        dcs6->setRot(((20 - (z- 42 *(i - 1)))/3.0),0 ,0);
      }

      dcs0->setTrans( X + 1.0 + z * 0.1  , Y , Z + 0.2 + 0.08f
```

```
            *cos(2.0f * 3.14159f * z / 21.0f));


        UpdateView();
        UpdateGUI();
        pfFrame();
    }
    for ( z = (i-1)*42 + 22  ; z < (i-1)*42 + 43 ; z += 0.5 * delta_t){
        /*  Go to sleep until next frame time. */
        pfSync();
        Shared->simTime = pfGetTime();

        /*    Main Body    */
        dcs0->setRot(90.0, 90.0 , 0);
        dcs0->setTrans(X,Y,Z);

        /*    Head    */
        dcs7->setTrans(0.0  , 0.4 , 0.0);

        /*    Torso    */
        dcs6->setScale(2.0f);
        dcs6->setTrans(0.3  , 2.1 , -0.005);
        dcs3->setScale(2.0f);
        dcs3->setTrans(0.3  , 2.1 , 0.005);

        /* Left foot    */
        dcs9->setScale(0.7f);
        dcs9->setRot(0, 0 , 180.0);
        dcs9->setTrans(0  , -1.7, 0);

        /* Right foot    */
        dcs8->setScale(0.7f);
        dcs8->setRot(0, 0 , 180.0);
        dcs8->setTrans(0  , -1.7, 0);

        /* Left lowerleg    */
        dcs4->setTrans(0  , -1.7, 0);

        /* Left upperleg    */
        dcs5->setTrans(0.6,  0,   0);

        /* Right lowerleg    */
        dcs1->setTrans(0  , -1.7, 0);

        /* Right upper arm    */
        dcs11->setScale(0.25f);
        dcs11->setTrans(-1.0 , 1.7, 0);

        /* Right lower arm    */
        dcs12->setTrans(0.4 , -4.6, 0.0);

        /* Right hand    */
        dcs13->setTrans(0.0 , -4.2, 0.0);
        dcs13->setRot(0, 0 , 180.0);
```

```
        /* Left upper arm    */
        dcs14->setScale(0.25f);
        dcs14->setTrans(1.6  , 1.8, 0);

        /* Left lower arm    */
        dcs15->setRot(0, 0 , 180.0);
        dcs15->setTrans(-0.5 , -4.6, 0.0);

        /* Left hand    */
        dcs16->setTrans(-0.2 , -4.6, 0.0);

////////////////////////////////////////////////////
//                Inverse Kinematics
////////////////////////////////////////////////////


////////////////////////////////////////////////////
//  SECOND STEP
//  RIGHT LEG STEP

        x_right = 0.95f * cos((64 - 2* (z- 42 *(i - 1))) * DEG_TO_RAD )
            * (l1 + l2) * cos((64 - 2* (z- 42 *(i - 1))) * DEG_TO_RAD);
        y_right = 0.95f * cos((64 - 2* (z- 42 *(i - 1))) * DEG_TO_RAD )
            * (l1 + l2) * sin((64 - 2* (z- 42 *(i - 1))) * DEG_TO_RAD);

        c1_right= ((x_right*x_right) + (y_right*y_right)
                - (l1*l1) - (l2*l2))/(2 * l1 *l2);
        theta2 = (1 *acos(c1_right));

        k1_right = l1 + (l2 * cos(theta2));
        k2_right = l2 * sin(theta2);

        theta1 = atan(y_right/x_right) - atan(k2_right/k1_right);

        //  Right Leg
        dcs2->setRot(0 ,(57.3 * theta1),0 );
        dcs1->setRot(0 ,(57.3 * theta2), 0);

        // Left ARM
        dcs14->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0);
        dcs15->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f),0);

//  LEFT LEG STEP
        x_left = cos((-64 + 2* (z- 42 *(i - 1))) * DEG_TO_RAD )
              * (l1 + l2) * cos((-64 + 2* (z- 42 *(i - 1))) * DEG_TO_RAD);
        y_left = cos((-64 + 2* (z- 42 *(i - 1))) * DEG_TO_RAD )
              * (l1 + l2) * sin((-64 + 2* (z- 42 *(i - 1))) * DEG_TO_RAD);

        c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
              - (l2*l2))/(2 * l1 *l2);
        theta2 = (1 *acos(c1_left));

        k1_left = l1 + (l2 * cos(theta2));
        k2_left = l2 * sin(theta2);
        theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);
```

```
        dcs5->setRot(0 ,(57.3 * theta1), 0);
        dcs4->setRot(0 ,(57.3 * theta2), 0);

        // Right arm
        dcs11->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
        dcs12->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);

        // Torso rotation
        if((z- 42 *(i - 1)) < 31){
          dcs3->setRot((-(z- 42 *(i - 1) - 21)/3.0),0 ,0);
          dcs6->setRot((-(z- 42 *(i - 1)  -21)/3.0),0 ,0);
        }
        else{
          dcs3->setRot(((-20 + ((z- 42 *(i - 1)- 21)))/3.0),0 ,0);
          dcs6->setRot(((-20 + ((z- 42 *(i - 1)- 21)))/3.0),0 ,0);
        }

        dcs0->setTrans(X + 1.0 + z * 0.1  , Y , Z + 0.2 + 0.08f
                    * cos(2.0f * 3.14159f * z / 21.0f));
        UpdateView();
        UpdateGUI();
        pfFrame();
      }
    }//Big FOR

//  LAST  HALF STEP /////////
    for ( z = 1 ; z < 12 ; z += 0.5 * delta_t){

        /*  Go to sleep until next frame time. */
        pfSync();
        Shared->simTime=pfGetTime();

        /*    Main Body    */

        dcs0->setRot(90.0, 90.0 , 0);
        dcs0->setTrans(X,Y,Z);

        /*    Head    */
        dcs7->setTrans(0.0  , 0.4 , 0.0);

        /*    Torso    */
        dcs6->setScale(2.0f);
        dcs6->setTrans(0.3  , 2.1 , -0.005);
        dcs3->setScale(2.0f);
        dcs3->setTrans(0.3  , 2.1 , 0.005);

        /* Left foot    */
        dcs9->setScale(0.7f);
        dcs9->setRot(0, 0 , 180.0);
        dcs9->setTrans(0  , -1.7, 0);

        /* Right foot    */
        dcs8->setScale(0.7f);
        dcs8->setRot(0, 0 , 180.0);
```

```
        dcs8->setTrans(0  , -1.7, 0);

        /* Left lowerleg   */
        dcs4->setTrans(0  , -1.7, 0);

        /* Left upperleg   */
        dcs5->setTrans(0.6,  0,   0);

        /* Right lowerleg   */
        dcs1->setTrans(0  , -1.7, 0);

        /* Right upper arm   */
        dcs11->setScale(0.25f);
        dcs11->setTrans(-1.0 , 1.7, 0);

        /* Right lower arm   */
        dcs12->setTrans(0.4 , -4.6, 0.0);

        /* Right hand   */
        dcs13->setTrans(0.0 , -4.2, 0.0);
        dcs13->setRot(0, 0 , 180.0);

        /* Left upper arm   */
        dcs14->setScale(0.25f);
        dcs14->setTrans(1.6  , 1.8, 0);

        /* Left lower arm   */
        dcs15->setRot(0, 0 , 180.0);
        dcs15->setTrans(-0.5 , -4.6, 0.0);

        /* Left hand   */
        dcs16->setTrans(-0.2 , -4.6, 0.0);

////////////////////////////////////////////////////
//              Inverse Kinematics
////////////////////////////////////////////////////
//   LEFT LEG

        x_left = 0.95f * cos((22 - 2* z) * DEG_TO_RAD ) *(l1 + l2)
                * cos((22 - 2* z) * DEG_TO_RAD);
        y_left = 0.95f * cos((22 - 2* z) * DEG_TO_RAD ) *(l1 + l2)
                * sin((22 - 2* z) * DEG_TO_RAD);

        c1_left= ((x_left*x_left) + (y_left*y_left)
                - (l1*l1) - (l2*l2))/(2 * l1 *l2);

        theta2 = (1 *acos(c1_left));

        k1_left = l1 + (l2 * cos(theta2));
        k2_left = l2 * sin(theta2);
        theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);

        dcs5->setRot(0 ,(57.3 * theta1),0 );
        dcs4->setRot(0 ,(57.3 * theta2) , 0);
```

```
        // Right arm
        dcs11->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
        dcs12->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);

        //   RIGHT LEG

        x_right = cos((-22 + 2 * z) * DEG_TO_RAD ) * (l1 + l2)
                * cos((-22 + 2 * z) * DEG_TO_RAD);
        y_right = cos((-22 + 2 * z) * DEG_TO_RAD ) * (l1 + l2)
                * sin((-22 + 2 * z) * DEG_TO_RAD);
        c1_right= ((x_right*x_right) + (y_right*y_right) - (l1*l1)
                - (l2*l2))/(2 * l1 *l2);
        theta2 = (1 *acos(c1_right));

        k1_right = l1 + (l2 * cos(theta2));
        k2_right = l2 * sin(theta2);
        theta1 = atan(y_right/x_right) - atan(k2_right/k1_right);

        dcs2->setRot(0 ,(57.3 * theta1), 0);
        dcs1->setRot(0 ,(57.3 * theta2), 0);

        // Left arm
        dcs14->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
        dcs15->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);
        // TORSO rotation
        if(z < 11){
          dcs3->setRot((z/3.0),0 ,0);
          dcs6->setRot((z/3.0),0 ,0);
        }
        else{
          dcs3->setRot(((20 - z)/3.0),0 ,0);
          dcs6->setRot(((20 - z)/3.0),0 ,0);
        }
        dcs0->setTrans(X + 1.0 + (i-1) * 4.2 + z * 0.1
              , Y , +Z + 0.2 + 0.08f *cos(2.0f * 3.14159f * z / 21.0f));

        UpdateView();
        UpdateGUI();
        pfFrame();
      }// END of step_forward
}
```

```
//-------------------------------------------------------------------------
// Function:      step_upward
// Returns:       None
// Parameters:    Number of steps needs to be taken, initial position
// Summary:       Computes the joint angles according to the position
//                of the end effector(foot) by using the inverse kinematic
//                equations of the three link planar manipulator. The
//                algorithm for the path of the foot is described in
//                Chapter IV of this thesis as "Stepping Upward Algorithm.
//                Height of each step is 0.267949 units.
//-------------------------------------------------------------------------

void step_upward(int number_of_steps,float X,float Y,float Z){


  float l1 = 1.0f;
  float l2 = 1.0f;
  float c1_left,c1_right;
  float theta1,theta2;
  float k1_left,k2_left;
  float k1_right,k2_right;

  float x_left =  1.732f;
  float y_left =  0.0f;
  float x_right =  1.732f;
  float y_right =  0.0f;
  float  z;

////////////////////////////////////
//   FIRST HALF STEP
////////////////////////////////////
  for ( z = 11.0 ; z < 22.0 ; z += 0.3 * delta_t)
  {

    /*  Go to sleep until next frame time. */
    pfSync();
    Shared->simTime=pfGetTime();

    /*   Main Body   */
    dcs0->setRot(90.0, 90.0 , 0.0);
    dcs0->setTrans(X,Y,Z);

    /*   Head   */
    dcs7->setTrans(0.0 , 0.4 , 0.0);

    /*   Torso   */
    dcs6->setScale(2.0f);
    dcs6->setTrans(0.3  , 2.1 , -0.005);
    dcs3->setScale(2.0f);
    dcs3->setTrans(0.3  , 2.1 , 0.005);


     /* Left foot   */
    dcs9->setScale(0.7f);
    dcs9->setRot(0, 0 , 180.0);
```

```
    dcs9->setTrans(0  , -1.7, 0);

    /* Right foot    */
    dcs8->setScale(0.7f);
    dcs8->setRot(0, 0 , 180.0);
    dcs8->setTrans(0  , -1.7, 0);

     /* Left lowerleg    */
    dcs4->setTrans(0  , -1.7, 0);

    /* Left upperleg    */
    dcs5->setTrans(0.6,  0,   0);

    /* Right lowerleg    */
    dcs1->setTrans(0  , -1.7, 0);

    /* Right upper arm    */
    dcs11->setScale(0.25f);

    /* Right lower arm    */
    dcs12->setTrans(0.4 , -4.6, 0.0);

    /* Right hand    */
    dcs13->setTrans(0.0 , -4.2, 0.0);
    dcs13->setRot(0, 0 , 180.0);

    /* Left upper arm    */
    dcs14->setScale(0.25f);
    //dcs14->setTrans(1.6  , 1.8, 0);

    /* Left lower arm    */
    dcs15->setRot(0, 0 , 180.0);
    dcs15->setTrans(-0.5 , -4.6, 0.0);

    /* Left hand    */
    dcs16->setTrans(-0.2 , -4.6, 0.0);

////////////////////////////////////////////////////////
//                Inverse Kinematics
////////////////////////////////////////////////////////
//  FIRST STEP
//   LEFT LEG
    x_left = 0.95f * cos((22 - 2* z - 10) * DEG_TO_RAD ) *(l1 + l2)
          * cos((22 - 2* z - 10) * DEG_TO_RAD);
    y_left = 0.95f * cos((22 - 2* z - 10) * DEG_TO_RAD ) *(l1 + l2)
          * sin((22 - 2* z - 10) * DEG_TO_RAD);

    c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
          - (l2*l2))/(2 * l1 *l2);

    theta2 = (1 *acos(c1_left));

    k1_left = l1 + (l2 * cos(theta2));
    k2_left = l2 * sin(theta2);
    theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);
```

```
    dcs5->setRot(0 ,(57.3 * theta1),0 );
    dcs4->setRot(0 ,(57.3 * theta2) , 0);

    // Right arm
    dcs11->setTrans(-1.0 , 1.7, 0.5);
    dcs11->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
    dcs12->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);

//  RIGHT LEG
    x_right = cos((-22 + 2 * z - 10) * DEG_TO_RAD ) * (l1 + l2)
          * cos((-22 + 2 * z - 10) * DEG_TO_RAD);
    y_right = cos((-22 + 2 * z - 10) * DEG_TO_RAD ) * (l1 + l2)
          * sin((-22 + 2 * z - 10) * DEG_TO_RAD);
    c1_right= ((x_right*x_right) + (y_right*y_right) - (l1*l1)
          - (l2*l2))/(2 * l1 *l2);
    theta2 = (1 *acos(c1_right));

    k1_right = l1 + (l2 * cos(theta2));
    k2_right = l2 * sin(theta2);
    theta1 = atan(y_right/x_right) - atan(k2_right/k1_right);

    dcs2->setRot(0 ,(57.3 * theta1), 0);
    dcs1->setRot(0 ,(57.3 * theta2), 0);
    // Left arm
    dcs14->setTrans(1.6  , 1.8, 0.5);
    dcs14->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
    dcs15->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);
    // TORSO rotation
    if(z < 11){
       dcs3->setTrans(0.3  , 2.1 , 0.305);
       dcs6->setTrans(0.3  , 2.1 , 0.305);
       dcs3->setRot((z/3.0),7.0,0);
       dcs6->setRot((z/3.0),7.0,0);
    }
    else{
       dcs3->setTrans(0.3  , 2.1 , 0.305);
       dcs6->setTrans(0.3  , 2.1 , 0.305);
       dcs3->setRot(((20 - z)/3.0) ,7.0,0);
       dcs6->setRot(((20 - z)/3.0) ,7.0,0);
    }
    z = z+21;
    dcs0->setTrans( X + (z - 32) * 0.1 -0.4, Y ,
        Z - 2.005063 + /* 0.08f *cos(2.0f * 3.14159f * z / 21.0f) */0
        + 1 * 0.267949 * 3.4 *2  + ((l1 + l2)
        -(l1 + l2) * cos(( -22 + 2 * z - 10) * DEG_TO_RAD)));
    z = z-21;


    UpdateView();
    UpdateGUI();
    pfFrame();
  }//END FOR
```

```
for( i = 1 ; i < number_of_steps ; i++){

   for ( z = (i-1)*42 + 22.0  ; z < (i-1)*42 + 43.0 ; z += 0.35 * delta_t){
      /*  Go to sleep until next frame time. */
      pfSync();
      Shared->simTime = pfGetTime();

      /*    Main Body    */
      dcs0->setRot(90.0, 90.0 , 0.0);
      dcs0->setTrans(X,Y,Z);

      /*    Head    */
      dcs7->setTrans(0.0 , 0.4 , 0.0);

      /*    Torso    */
      dcs6->setScale(2.0f);
      dcs6->setTrans(0.3  , 2.1 , -0.005);
      dcs3->setScale(2.0f);

      dcs3->setTrans(0.3  , 2.1 , 0.005);

      /* Left foot    */
      dcs9->setScale(0.7f);
      dcs9->setRot(0, 0 , 180.0);
      dcs9->setTrans(0  , -1.7, 0);

      /* Right foot    */
      dcs8->setScale(0.7f);
      dcs8->setRot(0, 0 , 180.0);
      dcs8->setTrans(0  , -1.7, 0);

      /* Left lowerleg    */
      dcs4->setTrans(0  , -1.7, 0);

      /* Left upperleg    */
      dcs5->setTrans(0.6,  0,   0);

      /* Right lowerleg    */
      dcs1->setTrans(0  , -1.7, 0);

      /* Right upper arm    */
      dcs11->setScale(0.25f);

      /* Right lower arm    */
      dcs12->setTrans(0.4 , -4.6, 0.0);

      /* Right hand    */
      dcs13->setTrans(0.0 , -4.2, 0.0);
      dcs13->setRot(0, 0 , 180.0);

      /* Left upper arm    */
      dcs14->setScale(0.25f);

      /* Left lower arm    */
      dcs15->setRot(0, 0 , 180.0);
```

```
        dcs15->setTrans(-0.5 , -4.6, 0.0);

        /* Left hand    */
        dcs16->setTrans(-0.2 , -4.6, 0.0);

////////////////////////////////////////////////////
//                Inverse Kinematics
////////////////////////////////////////////////////


////////////////////////////////////////////////////////
//  SECOND STEP
//   RIGHT LEG STEP

        x_right = 0.95f * cos((64 - 2* (z- 42 *(i - 1)) - 10) * DEG_TO_RAD)
                * (l1 + l2) * cos((64 - 2* (z- 42 *(i - 1)) - 10)
                * DEG_TO_RAD);
        y_right = 0.95f * cos((64 - 2* (z- 42 *(i - 1)) - 10) * DEG_TO_RAD)
                * (l1 + l2) * sin((64 - 2* (z- 42 *(i - 1)) - 10)
                * DEG_TO_RAD);
        c1_right= ((x_right*x_right) + (y_right*y_right) - (l1*l1)
                 - (l2*l2))/(2 * l1 *l2);
        theta2 = (1 *acos(c1_right));

        k1_right = l1 + (l2 * cos(theta2));
        k2_right = l2 * sin(theta2);
        theta1 = atan(y_right/x_right) - atan(k2_right/k1_right);

        //  Right LEG
        dcs2->setRot(0 ,(57.3 * theta1),0 );
        dcs1->setRot(0 ,(57.3 * theta2), 0);

        // Left arm
        dcs14->setTrans(1.6  , 1.8, 0.5);
        dcs14->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
        dcs15->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);


        //    LEFT LEG STEP
        x_left = cos((-64 + 2* (z- 42 *(i - 1)) - 10) * DEG_TO_RAD )
                * (l1 + l2)
                * cos((-64 + 2* (z- 42 *(i - 1)) - 10) * DEG_TO_RAD);
        y_left = cos((-64 + 2* (z- 42 *(i - 1)) - 10) * DEG_TO_RAD )
                * (l1 + l2)
                * sin((-64 + 2* (z- 42 *(i - 1)) - 10) * DEG_TO_RAD);

        c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
                 - (l2*l2))/(2 * l1 *l2);
        theta2 = (1 *acos(c1_left));

        k1_left = l1 + (l2 * cos(theta2));
        k2_left = l2 * sin(theta2);
        theta1 = atan(y_left/x_left) - atan(k2_left/k1_left):

        dcs5->setRot(0 ,(57.3 * theta1), 0);
```

```
        dcs4->setRot(0 ,(57.3 * theta2), 0);

        // Right arm
        dcs11->setTrans(-1.0 , 1.7, 0.5);
        dcs11->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
        dcs12->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);

       // TORSO rotation
       if((z- 42 *(i - 1)) < 31){
          dcs3->setTrans(0.3  , 2.1 , 0.305);
          dcs6->setTrans(0.3  , 2.1 , 0.305);
          dcs3->setRot((-(z- 42 *(i - 1) - 21)/3.0) ,7.0,0);
          dcs6->setRot((-(z- 42 *(i - 1)  -21)/3.0) ,7.0,0);
       }
       else{
          dcs3->setTrans(0.3  , 2.1 , 0.305);
      dcs6->setTrans(0.3  , 2.1 , 0.305);
      dcs3->setRot(((-20 + ((z- 42 *(i - 1)- 21)))/3.0) ,7.0,0);
      dcs6->setRot(((-20 + ((z- 42 *(i - 1)- 21)))/3.0) ,7.0,0);

      }

      z = z-21;
      dcs0->setTrans( X + 0.5 + z * 0.1  , Y , Z
         + i * 0.267949 * 3.4 *2 - ((l1 + l2) - (l1 + l2)
         * cos(( 64 - 2 * (z- 42 *(i - 1)) ) * DEG_TO_RAD)));
      z = z+21;

      UpdateView();
      UpdateGUI();
      pfFrame();
   }

for ( z = (i-1)*42 + 1.0 ; z < (i-1)*42 + 22.0 ; z += 0.35 * delta_t){

      /*  Go to sleep until next frame time. */
      pfSync();
      Shared->simTime=pfGetTime();

      /*    Main Body    */

      dcs0->setRot(90.0, 90.0 ,0);
      dcs0->setTrans(X,Y,Z);

      /*    Head    */
      dcs7->setTrans(0.0  , 0.4 , 0.0);

      /*    Torso    */
      dcs6->setScale(2.0f);
      dcs6->setTrans(0.3  , 2.1 , -0.005);
      dcs3->setScale(2.0f);
      dcs3->setTrans(0.3  , 2.1 , 0.005);
```

```
        /* Left foot    */
        dcs9->setScale(0.7f);
        dcs9->setRot(0, 0 , 180.0);
        dcs9->setTrans(0  , -1.7, 0);

        /* Right foot    */
        dcs8->setScale(0.7f);
        dcs8->setRot(0, 0 , 180.0);
        dcs8->setTrans(0  , -1.7, 0);

        /* Left lowerleg    */
        dcs4->setTrans(0  , -1.7, 0);

        /* Left upperleg    */
        dcs5->setTrans(0.6,  0,   0);

        /* Right lowerleg    */
        dcs1->setTrans(0  , -1.7, 0);

        /* Right upper arm    */
        dcs11->setScale(0.25f);

        /* Right lower arm    */
        dcs12->setTrans(0.4 , -4.6, 0.0);

        /* Right hand    */
        dcs13->setTrans(0.0 , -4.2, 0.0);
        dcs13->setRot(0, 0 , 180.0);

        /* Left upper arm    */
        dcs14->setScale(0.25f);

        /* Left lower arm    */
        dcs15->setRot(0, 0 , 180.0);
        dcs15->setTrans(-0.5 , -4.6, 0.0);

        /* Left hand    */
        dcs16->setTrans(-0.2 , -4.6, 0.0);

////////////////////////////////////////////////////////
//                  Inverse Kinematics
////////////////////////////////////////////////////////
//  FIRST STEP
//   LEFT LEG
        x_left = 0.95f * cos((22 - 2* (z- 42 *(i - 1)) - 10) * DEG_TO_RAD )
                *(l1 + l2) * cos((22 - 2* (z- 42 *(i - 1)) - 10) *DEG_TO_RAD);
        y_left = 0.95f * cos((22 - 2* (z- 42 *(i - 1)) - 10) * DEG_TO_RAD )
                *(l1 + l2) * sin((22 - 2* (z- 42 *(i - 1)) - 10) *DEG_TO_RAD);
        c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
                - (l2*l2))/(2 * l1 *l2);

        theta2 = (1 *acos(c1_left));

        k1_left = l1 + (l2 * cos(theta2));
        k2_left = l2 * sin(theta2);
```

```
      theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);

      dcs5->setRot(0 ,(57.3 * theta1),0 );
      dcs4->setRot(0 ,(57.3 * theta2) , 0);

      // Right arm
      dcs11->setTrans(-1.0 , 1.7, 0.5);
      dcs11->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
      dcs12->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);

//    RIGHT LEG
      x_right = cos((-22 + 2 * (z- 42 *(i - 1)) - 10) * DEG_TO_RAD )
            * (l1 + l2) * cos((-22 + 2 * (z- 42 *(i - 1)) - 10)
            * DEG_TO_RAD);
      y_right = cos((-22 + 2 * (z- 42 *(i - 1)) - 10) * DEG_TO_RAD )
            * (l1 + l2) * sin((-22 + 2 * (z- 42 *(i - 1)) - 10)
            * DEG_TO_RAD);
      c1_right= ((x_right*x_right) + (y_right*y_right) - (l1*l1)
            - (l2*l2))/(2 * l1 *l2);
      theta2 = (1 *acos(c1_right));

      k1_right = l1 + (l2 * cos(theta2));
      k2_right = l2 * sin(theta2);
      theta1 = atan(y_right/x_right) - atan(k2_right/k1_right);

      dcs2->setRot(0 ,(57.3 * theta1), 0);
      dcs1->setRot(0 ,(57.3 * theta2), 0);
      // Left arm
      dcs14->setTrans(1.6  , 1.8, 0.5);
      dcs14->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
      dcs15->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);
      // TORSO rotation
      if((z- 42 *(i - 1)) < l1){
         dcs3->setTrans(0.3  , 2.1 , 0.305);
         dcs6->setTrans(0.3  , 2.1 , 0.305);
         dcs3->setRot(((z- 42 *(i - 1))/3.0) ,7.0 ,0);
         dcs6->setRot(((z- 42 *(i - 1))/3.0) ,7.0 ,0);
      }
      else{
         dcs3->setTrans(0.3  , 2.1 , 0.305);
         dcs6->setTrans(0.3  , 2.1 , 0.305);
         dcs3->setRot(((20 - (z- 42 *(i - 1)))/3.0) ,7.0 ,0);
         dcs6->setRot(((20 - (z- 42 *(i - 1)))/3.0) ,7.0 ,0);
      }
      z = z+21;
      dcs0->setTrans(X + 0.5 + z * 0.1  , Y , Z -0.105327
                  + i * 0.267949 * 3.4 *2  + ((l1 + l2) -(l1 + l2)
                  * cos(( -22 + 2 * (z- 42 *(i - 1)) - 10)
                  * DEG_TO_RAD)));
       z = z-21;
      UpdateView();
      UpdateGUI();
      pfFrame();
   }//END FOR
}//Big FOR
```

```
////////////////////////////////////////////////
//              LAST HALF
////////////////////////////////////////////////

 for ( z = 22.0  ; z < 33.0 ; z += 0.35 * delta_t){
    /*  Go to sleep until next frame time. */
    pfSync();
    Shared->simTime = pfGetTime();

    /*    Main Body    */
    dcs0->setRot(90.0, 90.0 , 0.0);
    dcs0->setTrans(X,Y,Z);

    /*    Head    */
    dcs7->setTrans(0.0 , 0.4 , 0.0);

    /*    Torso    */
    dcs6->setScale(2.0f);
    dcs6->setTrans(0.3  , 2.1 , -0.005);
    dcs3->setScale(2.0f);
    dcs3->setTrans(0.3  , 2.1 , 0.005);


    /* Left foot    */
    dcs9->setScale(0.7f);
    dcs9->setRot(0, 0 , 180.0);
    dcs9->setTrans(0  , -1.7, 0);

    /* Right foot    */
    dcs8->setScale(0.7f);
    dcs8->setRot(0, 0 , 180.0);
    dcs8->setTrans(0  , -1.7, 0);

    /* Left lowerleg    */
    dcs4->setTrans(0  , -1.7, 0);

    /* Left upperleg    */
    dcs5->setTrans(0.6,  0,   0);

    /* Right lowerleg    */
    dcs1->setTrans(0  , -1.7, 0);

    /* Right upper arm    */
    dcs11->setScale(0.25f);

    /* Right lower arm    */
      dcs12->setTrans(0.4 , -4.6, 0.0);

    /* Right hand    */
    dcs13->setTrans(0.0 , -4.2, 0.0);
    dcs13->setRot(0, 0 , 180.0);

    /* Left upper arm    */
    dcs14->setScale(0.25f);
```

```
    /* Left lower arm   */
    dcs15->setRot(0, 0 , 180.0);
    dcs15->setTrans(-0.5 , -4.6, 0.0);

    /* Left hand   */
    dcs16->setTrans(-0.2 , -4.6, 0.0);

////////////////////////////////////////////////////
//              Inverse Kinematics
////////////////////////////////////////////////////


////////////////////////////////////////////////////
//  SECOND STEP
//  RIGHT LEG STEP

    x_right = 0.95f * cos((64 - 2* z - 10) * DEG_TO_RAD )* (l1 + l2)
            * cos((64 - 2* z - 10) * DEG_TO_RAD);
    y_right = 0.95f * cos((64 - 2* z - 10) * DEG_TO_RAD )* (l1 + l2)
            * sin((64 - 2* z - 10) * DEG_TO_RAD);

    c1_right= ((x_right*x_right) + (y_right*y_right) - (l1*l1)
            - (l2*l2))/(2 * l1 *l2);
    theta2 = (1 *acos(c1_right));

    k1_right = l1 + (l2 * cos(theta2));
    k2_right = l2 * sin(theta2);

    theta1 = atan(y_right/x_right) - atan(k2_right/k1_right);

    //  Right LEG
    dcs2->setRot(0 ,(57.3 * theta1),0 );
    dcs1->setRot(0 ,(57.3 * theta2), 0);

    // Left arm
    dcs14->setTrans(1.6  , 1.8, 0.5);
    dcs14->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
    dcs15->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);

//   LEFT LEG STEP
    x_left = cos((-64 + 2* z - 10) * DEG_TO_RAD ) * (l1 + l2)
           * cos((-64 + 2* z - 10) * DEG_TO_RAD);
    y_left = cos((-64 + 2* z - 10) * DEG_TO_RAD ) * (l1 + l2)
           * sin((-64 + 2* z - 10) * DEG_TO_RAD);

    c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
           - (l2*l2))/(2 * l1 *l2);
    theta2 = (1 *acos(c1_left));


    k1_left = l1 + (l2 * cos(theta2));
    k2_left = l2 * sin(theta2);
    theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);

    dcs5->setRot(0 ,(57.3 * theta1), 0);
```

```
    dcs4->setRot(0 ,(57.3 * theta2), 0);
    // Right arm

    dcs11->setTrans(-1.0 , 1.7, 0.5);
    dcs11->setRot(0 ,(0.7f *(57.3 * theta1) + 10.0f),0 );
    dcs12->setRot(0 ,(1.5f *(57.3 * theta1) - 10.0f), 0);

    // Torso rotation
    if(z < 31){
       dcs3->setTrans(0.3  , 2.1 , 0.305);
       dcs6->setTrans(0.3  , 2.1 , 0.305);
       dcs3->setRot((-z - 21)/3.0 +13.75,7.0,0);
       dcs6->setRot((-z  -21)/3.0 +13.75,7.0,0);
    }
    else{
       dcs3->setTrans(0.3  , 2.1 , 0.305);
       dcs6->setTrans(0.3  , 2.1 , 0.305);
       dcs3->setRot((-20+ z- 21)/3.0 ,7.0,0);
       dcs6->setRot((-20 + z- 21)/3.0 ,7.0,0);
    }
    z = z-21;
    dcs0->setTrans( X + 0.5 +(i-1) * 4.2+ z * 0.1  , Y , Z
                + number_of_steps * 0.267949 * 3.4 *2 - ((l1 + l2)
                - (l1 + l2) * cos(( 64 - 2 * z ) * DEG_TO_RAD)));

    z = z+21;

    UpdateView();
    UpdateGUI();
    pfFrame();
    }//End of for
}//End of step_upward
```

```
//-----------------------------------------------------------------------
//  Function:       jump
//  Returns:        None
//  Parameters:     Initial position
//  Summary:        Translate the whole body first straight upward,
//                  secondly along a semi circle path, then straight down
//                  and straight up to an upright position. While translating
//                  the body appropriate joint angles are applied.
//-----------------------------------------------------------------------

void
jump(float X,float Y,float Z){

   float l1 = 1.0f;
   float l2 = 1.0f;
   float c1_left,c1_right;
   float theta1,theta2;
   float k1_left,k2_left;
   float k1_right,k2_right;

   float x_left =  1.732f;
   float y_left =  0.0f;
   float x_right =  1.732f;
   float y_right =  0.0f;

   for (float  j = 1  ; j < 101 ; j += delta_t){
      /*  Go to sleep until next frame time. */
      pfSync();
      Shared->simTime = pfGetTime();

      /*    Main Body    */
      dcs0->setRot(90.0, 90.0 , 0);
      dcs0->setTrans(X+j*0.8/100.0,Y,Z - j * 1.5/100.0);

      /*    Head    */
      dcs7->setTrans(0.0  , 0.4 , 0.0);

      /*    Torso    */
      dcs6->setScale(2.0f);
      dcs6->setTrans(0.3  , 2.1 , -0.005);
      dcs3->setScale(2.0f);
      dcs3->setTrans(0.3  , 2.1 , 0.005);
      dcs3->setRot(0, 10.0*j/100.0 , 0.0);
      dcs6->setRot(0, 10.0*j/100.0 , 0.0);

      x_left = 1.95 - 1.0 *j/100.0;
      y_left = 0.0;

      c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
            - (l2*l2))/(2 * l1 *l2);

      theta2 = (1 *acos(c1_left));

      k1_left = l1 + (l2 * cos(theta2));
      k2_left = l2 * sin(theta2);
```

```
    theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);

    dcs5->setRot(0 ,(57.3 * theta1),0 );
    dcs4->setRot(0 ,(57.3 * theta2) , 0);
    dcs2->setRot(0 ,(57.3 * theta1),0 );
    dcs1->setRot(0 ,(57.3 * theta2) , 0);

    /* Left foot   */
    dcs9->setScale(0.7f);
    dcs9->setRot(0, 0 , 180.0);
    dcs9->setTrans(0  , -1.7, 0);

    /* Right foot   */
    dcs8->setScale(0.7f);
    dcs8->setRot(0, 0 , 180.0);
    dcs8->setTrans(0  , -1.7, 0);

    /* Left lowerleg   */
    dcs4->setTrans(0  , -1.7, 0);

    /* Left upperleg   */
    dcs5->setTrans(0.6,  0,   0);

    /* Right lowerleg   */
    dcs1->setTrans(0  , -1.7, 0);

    /* Right upper arm   */
    dcs11->setScale(0.25f);
    dcs11->setTrans(-1.0 , 1.7, 0);
    dcs11->setRot(0.0,-j/5.0 ,0.0);

    /* Right lower arm   */
    dcs12->setTrans(0.4 , -4.6, 0.0);
    dcs12->setRot(0.0 ,-j/5.0,0.0);

    /* Right hand   */
    dcs13->setTrans(0.0 , -4.2, 0.0);
    dcs13->setRot(0, 0 , 180.0);

    /* Left upper arm   */
    dcs14->setScale(0.25f);
    dcs14->setTrans(1.6  , 1.8, 0);
    dcs14->setRot(0.0 ,-j/5.0,0.0);

    /* Left lower arm   */
    dcs15->setRot(0,-j/5.0, 180.0);
    dcs15->setTrans(-0.5 , -4.6, 0.0);

    /* Left hand   */
    dcs16->setTrans(-0.2 , -4.6, 0.0);

    UpdateView();
    UpdateGUI();
    pfFrame();
}
```

```
for (float  k = 1  ; k < 101 ; k += 5.0 * delta_t){
   /*  Go to sleep until next frame time. */
   pfSync();
   Shared->simTime = pfGetTime();

   x_left = 0.95 + 0.5 * k /100.0;
   y_left = 0.0;

   /*    Main Body    */
   dcs0->setRot(90.0 , 90.0 + 15.0*k/100.0 , 0);
   dcs0->setTrans(X+ 100.0*0.8/100.0 + (x_left* 1.7
              * sin(15.0*k*DEG_TO_RAD/100.0)), Y ,
              Z- 100.0 * 1.5/100.0 + 0.95 * k /100.0 - x_left* 1.7
             *(1.0 - cos(15.0*k*DEG_TO_RAD/100.0)));

   /*    Head    */
   dcs7->setTrans(0.0  , 0.4 , 0.0);

   /*    Torso    */
   dcs6->setScale(2.0f);
   dcs6->setTrans(0.3  , 2.1 , -0.005);
   dcs3->setScale(2.0f);
   dcs3->setTrans(0.3  , 2.1 , 0.005);
   dcs3->setRot(0, 10.0 , 0.0);
   dcs6->setRot(0, 10.0 , 0.0);

   c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
          - (l2*l2))/(2 * l1 *l2);

   theta2 = (1 *acos(c1_left));

   k1_left = l1 + (l2 * cos(theta2));
   k2_left = l2 * sin(theta2);
   theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);

   dcs5->setRot(0 ,(57.3 *  theta1) ,0);
   dcs4->setRot(0 ,(57.3 *  theta2) ,0);
   dcs2->setRot(0 ,(57.3 *  theta1) ,0);
   dcs1->setRot(0 ,(57.3 *  theta2) ,0);

   /* Left foot    */
   dcs9->setScale(0.7f);
   dcs9->setRot(0, 0 , 180.0);
   dcs9->setTrans(0  , -1.7, 0);

   /* Right foot    */
   dcs8->setScale(0.7f);
   dcs8->setRot(0, 0 , 180.0);
   dcs8->setTrans(0  , -1.7, 0);

   /* Left lowerleg    */
   dcs4->setTrans(0  , -1.7, 0);

   /* Left upperleg    */
   dcs5->setTrans(0.6,  0,   0);
```

```
    /* Right lowerleg   */
    dcs1->setTrans(0  , -1.7, 0);

    /* Right upper arm   */
    dcs11->setScale(0.25f);
    dcs11->setTrans(-1.0 , 1.7, 0);
    dcs11->setRot(0.0,-j/5.0 ,0.0);

    /* Right lower arm   */
    dcs12->setTrans(0.4 , -4.6, 0.0);
    dcs12->setRot(0.0 ,-j/5.0,0.0);

    /* Right hand   */
    dcs13->setTrans(0.0 , -4.2, 0.0);
    dcs13->setRot(0, 0 , 180.0);

    /* Left upper arm   */
    dcs14->setScale(0.25f);
    dcs14->setTrans(1.6  , 1.8, 0);
    dcs14->setRot(0.0 ,-j/5.0,0.0);

    /* Left lower arm   */
    dcs15->setRot(0,-j/5.0, 180.0);
    dcs15->setTrans(-0.5 , -4.6, 0.0);

    /* Left hand   */
    dcs16->setTrans(-0.2 , -4.6, 0.0);

    UpdateView();
    UpdateGUI();
    pfFrame();
}
for (float  l = 1  ; l < 101 ; l += delta_t){
   /*  Go to sleep until next frame time. */
    pfSync();
    Shared->simTime = pfGetTime();

    x_left = 1.45;
    y_left = 0.0;

    /*   Main Body   */
    dcs0->setRot(90.0 , 90.0+ 360.0*l/100.0 , 0);

    dcs0->setTrans(3.0 + 3.0*cos((180.0 - 180.0* l/100.0)*DEG_TO_RAD)+ X
              + 100.0*0.8/100.0  + (x_left* 1.7
              * sin(15.0*100.0*DEG_TO_RAD/100.0)),Y ,
              3.0*sin((180.0 - 180.0* l/100.0)*DEG_TO_RAD)
              + Z- 100.0 * 1.5/100.0+ 0.95 * 100.0/100.0 - x_left
              * 1.7*(1.0 - cos(15.0*100.0*DEG_TO_RAD/100.0)));

    /*   Head   */
    dcs7->setTrans(0.0  , 0.4 , 0.0);
```

```
/*   Torso   */
dcs6->setScale(2.0f);
dcs6->setTrans(0.3  , 2.1 , -0.005);
dcs3->setScale(2.0f);
dcs3->setTrans(0.3  , 2.1 , 0.005);
dcs3->setRot(0, 10.0 , 0.0);
dcs6->setRot(0, 10.0 , 0.0);

c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
        - (l2*l2))/(2 * l1 *l2);

theta2 = (1 *acos(c1_left));

k1_left = l1 + (l2 * cos(theta2));
k2_left = l2 * sin(theta2);
theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);

dcs5->setRot(0 ,(57.3 *  theta1) ,0);
dcs4->setRot(0 ,(57.3 *  theta2) ,0);
dcs2->setRot(0 ,(57.3 *  theta1) ,0);
dcs1->setRot(0 ,(57.3 *  theta2) ,0);

/* Left foot   */
dcs9->setScale(0.7f);
dcs9->setRot(0, 0 , 180.0);
dcs9->setTrans(0  , -1.7, 0);

/* Right foot   */
dcs8->setScale(0.7f);
dcs8->setRot(0, 0 , 180.0);
dcs8->setTrans(0  , -1.7, 0);

/* Left lowerleg   */
dcs4->setTrans(0  , -1.7, 0);

/* Left upperleg   */
dcs5->setTrans(0.6,  0,   0);

/* Right lowerleg   */
dcs1->setTrans(0  , -1.7, 0);

/* Right upper arm   */
dcs11->setScale(0.25f);
dcs11->setTrans(-1.0 , 1.7, 0);
dcs11->setRot(0.0,-100.0/5.0 ,0.0);

/* Right lower arm   */
dcs12->setTrans(0.4 , -4.6, 0.0);
dcs12->setRot(0.0 ,-100.0/5.0,0.0);

/* Right hand   */
dcs13->setTrans(0.0 , -4.2, 0.0);
dcs13->setRot(0, 0 , 180.0);
```

```
    /* Left upper arm    */
    dcs14->setScale(0.25f);
    dcs14->setTrans(1.6  , 1.8, 0);
    dcs14->setRot(0.0 ,-100.0/5.0,0.0);

    /* Left lower arm    */
    dcs15->setRot(0,-100.0/5.0, 180.0);
    dcs15->setTrans(-0.5 , -4.6, 0.0);

    /* Left hand    */
    dcs16->setTrans(-0.2 , -4.6, 0.0);

    UpdateView();
    UpdateGUI();
    pfFrame();
}
for (float  m = 1  ; m < 101 ; m += 3.0 * delta_t){
   /*  Go to sleep until next frame time. */
    pfSync();
    Shared->simTime = pfGetTime();

    x_left = 1.45;
    y_left = 0.0;

    /*    Main Body    */
    dcs0->setRot(90.0 , 90.0+ 360.0*100.0/100.0 , 0);
    dcs0->setTrans(3.0 + 3.0*cos((180.0 - 180.0* 100.0/100.0)*DEG_TO_RAD)+X
              + 100.0*0.8/100.0 + (x_left* 1.7
              * sin(15.0*100.0*DEG_TO_RAD/100.0)),Y ,
              3.0*sin((180.0 - 180.0* 100.0/100.0)*DEG_TO_RAD)
              + Z- 100.0 * 1.5/100.0 + 0.95 * 100.0/100.0 - x_left
              * 1.7*(1.0 - cos(15.0*100.0*DEG_TO_RAD/100.0))
              - 9.01*m/100.0);

    /*    Head    */
    dcs7->setTrans(0.0  , 0.4 , 0.0);

    /*    Torso    */
    dcs6->setScale(2.0f);
    dcs6->setTrans(0.3  , 2.1 , -0.005);
    dcs3->setScale(2.0f);
    dcs3->setTrans(0.3  , 2.1 , 0.005);
    dcs3->setRot(0, 10.0 , 0.0);
    dcs6->setRot(0, 10.0 , 0.0);

    c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
          - (l2*l2))/(2 * l1 *l2);

    theta2 = (1 *acos(c1_left));

    k1_left = l1 + (l2 * cos(theta2));
    k2_left = l2 * sin(theta2);
    theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);

    dcs5->setRot(0 ,(57.3 *  theta1) ,0);
```

```
    dcs4->setRot(0 ,(57.3 *  theta2) ,0);
    dcs2->setRot(0 ,(57.3 *  theta1) ,0);
    dcs1->setRot(0 ,(57.3 *  theta2) ,0);

    /* Left foot    */
    dcs9->setScale(0.7f);
    dcs9->setRot(0, 0 , 180.0);
    dcs9->setTrans(0  , -1.7, 0);

    /* Right foot    */
    dcs8->setScale(0.7f);
    dcs8->setRot(0, 0 , 180.0);
    dcs8->setTrans(0  , -1.7, 0);

    /* Left lowerleg    */
    dcs4->setTrans(0  , -1.7, 0);

    /* Left upperleg    */
    dcs5->setTrans(0.6,  0,   0);

    /* Right lowerleg    */
    dcs1->setTrans(0  , -1.7, 0);

    /* Right upper arm    */
    dcs11->setScale(0.25f);
    dcs11->setTrans(-1.0 , 1.7, 0);
    dcs11->setRot(0.0,-100.0/5.0 ,0.0);

    /* Right lower arm    */
    dcs12->setTrans(0.4 , -4.6, 0.0);
    dcs12->setRot(0.0 ,-100.0/5.0,0.0);

    /* Right hand    */
    dcs13->setTrans(0.0 , -4.2, 0.0);
    dcs13->setRot(0, 0 , 180.0);

    /* Left upper arm    */
    dcs14->setScale(0.25f);
    dcs14->setTrans(1.6  , 1.8, 0);
    dcs14->setRot(0.0 ,-100.0/5.0,0.0);

    /* Left lower arm    */
    dcs15->setRot(0,-100.0/5.0, 180.0);
    dcs15->setTrans(-0.5 , -4.6, 0.0);

    /* Left hand    */
    dcs16->setTrans(-0.2 , -4.6, 0.0);

    UpdateView();
    UpdateGUI();
    pfFrame();
}
for (float  n = 1  ; n < 101 ; n += 5.0 * delta_t){
   /*  Go to sleep until next frame time. */
    pfSync();
```

```
Shared->simTime = pfGetTime();

x_left = 1.45 + 0.5 *n/100.0;
y_left = 0.0;

/*   Main Body   */
dcs0->setRot(90.0 , 90.0+ 360.0*100.0/100.0 , 0);

dcs0->setTrans(3.0 + 3.0*cos((180.0 - 180.0* 100.0/100.0)*DEG_TO_RAD)+X
        + 100.0*0.8/100.0 + (1.45* 1.7
        * sin(15.0*100.0*DEG_TO_RAD/100.0)),Y ,
        3.0*sin((180.0 - 180.0* 100.0/100.0)*DEG_TO_RAD)
        + Z- 100.0 * 1.5/100.0 + 0.95 * 100.0/100.0 - 1.45
        * 1.7*(1.0 - cos(15.0*100.0*DEG_TO_RAD/100.0))
        -9.01*100.0/100.0+ n* 1.8/100.0);

/*   Head   */
dcs7->setTrans(0.0  , 0.4 , 0.0);

/*   Torso   */
dcs6->setScale(2.0f);
dcs6->setTrans(0.3  , 2.1 , -0.005);
dcs3->setScale(2.0f);
dcs3->setTrans(0.3  , 2.1 , 0.005);
dcs3->setRot(0, 10.0 , 0.0);
dcs6->setRot(0, 10.0 , 0.0);

c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
      - (l2*l2))/(2 * l1 *l2);

theta2 = (1 *acos(c1_left));

k1_left = l1 + (l2 * cos(theta2));
k2_left = l2 * sin(theta2);
theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);

dcs5->setRot(0 ,(57.3 *  theta1) ,0);
dcs4->setRot(0 ,(57.3 *  theta2) ,0);
dcs2->setRot(0 ,(57.3 *  theta1) ,0);
dcs1->setRot(0 ,(57.3 *  theta2) ,0);

/* Left foot   */
dcs9->setScale(0.7f);
dcs9->setRot(0, 0 , 180.0);
dcs9->setTrans(0  , -1.7, 0);

/* Right foot   */
dcs8->setScale(0.7f);
dcs8->setRot(0, 0 , 180.0);
dcs8->setTrans(0  , -1.7, 0);

/* Left lowerleg   */
dcs4->setTrans(0  , -1.7, 0);

/* Left-upperleg   */
```

```
    dcs5->setTrans(0.6,  0,   0);

    /* Right lowerleg    */
    dcs1->setTrans(0  , -1.7, 0);

    /* Right upper arm    */
    dcs11->setScale(0.25f);
    dcs11->setTrans(-1.0 , 1.7, 0);
    dcs11->setRot(0.0,-100.0/5.0 ,0.0);

    /* Right lower arm    */
    dcs12->setTrans(0.4 , -4.6, 0.0);
    dcs12->setRot(0.0 ,-100.0/5.0,0.0);

    /* Right hand    */
    dcs13->setTrans(0.0 , -4.2, 0.0);
    dcs13->setRot(0, 0 , 180.0);

    /* Left upper arm    */
    dcs14->setScale(0.25f);
    dcs14->setTrans(1.6  , 1.8, 0);
    dcs14->setRot(0.0 ,-100.0/5.0,0.0);

    /* Left lower arm    */
    dcs15->setRot(0,-100.0/5.0, 180.0);
    dcs15->setTrans(-0.5 , -4.6, 0.0);

    /* Left hand    */
    dcs16->setTrans(-0.2 , -4.6, 0.0);

    UpdateView();
    UpdateGUI();
    pfFrame();
}
for (float  o = 1  ; o < 101 ; o += 2.0 * delta_t){
   /*  Go to sleep until next frame time. */
    pfSync();
    Shared->simTime = pfGetTime();

    x_left = 1.45 + 0.5 *100.0/100.0;
    y_left = 0.0;

    /*    Main Body    */
    dcs0->setRot(90.0 , 90.0+ 360.0*100.0/100.0 , 0);

    dcs0->setTrans(3.0 + 3.0*cos((180.0 - 180.0* 100.0/100.0)*DEG_TO_RAD)+X
              + 100.0*0.8/100.0 + (1.45* 1.7
              * sin(15.0*100.0*DEG_TO_RAD/100.0)),Y , 3.0
              *sin((180.0 - 180.0* 100.0/100.0)*DEG_TO_RAD)
              + Z- 100.0 * 1.5/100.0 + 0.95 * 100.0/100.0 - 1.45* 1.7
              * (1.0 - cos(15.0*100.0*DEG_TO_RAD/100.0))
              -9.01*100.0/100.0 + n* 1.8/100.0);

    /*    Head    */
    dcs7->setTrans(0.0  , 0.4 , 0.0);
```

```
/*   Torso   */
dcs6->setScale(2.0f);
dcs6->setTrans(0.3  , 2.1 , -0.005);
dcs3->setScale(2.0f);
dcs3->setTrans(0.3  , 2.1 , 0.005);
dcs3->setRot(0, 10.0 , 0.0);
dcs6->setRot(0, 10.0 , 0.0);

c1_left= ((x_left*x_left) + (y_left*y_left) - (l1*l1)
        - (l2*l2))/(2 * l1 *l2);

theta2 = (1 *acos(c1_left));

k1_left = l1 + (l2 * cos(theta2));
k2_left = l2 * sin(theta2);
theta1 = atan(y_left/x_left) - atan(k2_left/k1_left);

dcs5->setRot(0 ,(57.3 *  theta1) ,0);
dcs4->setRot(0 ,(57.3 *  theta2) ,0);
dcs2->setRot(0 ,(57.3 *  theta1) ,0);
dcs1->setRot(0 ,(57.3 *  theta2) ,0);

/* Left foot   */
dcs9->setScale(0.7f);
dcs9->setRot(0, 0 , 180.0);
dcs9->setTrans(0  , -1.7, 0);

/* Right foot   */
dcs8->setScale(0.7f);
dcs8->setRot(0, 0 , 180.0);
dcs8->setTrans(0  , -1.7, 0);

/* Left lowerleg   */
dcs4->setTrans(0  , -1.7, 0);

/* Left upperleg   */
dcs5->setTrans(0.6,  0,   0);

/* Right lowerleg   */
dcs1->setTrans(0  , -1.7, 0);

/* Right upper arm   */
dcs11->setScale(0.25f);
dcs11->setTrans(-1.0 , 1.7, 0);
dcs11->setRot(0.0,-100.0/5.0 ,0.0);

/* Right lower arm   */
dcs12->setTrans(0.4 , -4.6, 0.0);
dcs12->setRot(0.0 ,-100.0/5.0,0.0);

/* Right hand   */
dcs13->setTrans(0.0 , -4.2, 0.0);
dcs13->setRot(0, 0 , 180.0);

/* Left upper arm   */
```

```
        dcs14->setScale(0.25f);
        dcs14->setTrans(1.6  , 1.8, 0);
        dcs14->setRot(0.0 ,-100.0/5.0,0.0);

        /* Left lower arm    */
        dcs15->setRot(0,-100.0/5.0, 180.0);
        dcs15->setTrans(-0.5 , -4.6, 0.0);

        /* Left hand    */
        dcs16->setTrans(-0.2 , -4.6, 0.0);

        UpdateView();
        UpdateGUI();
        pfFrame();
    }

}
```

LIST OF REFERENCES

[CAMA77] Camana, P. C., Hemami H., Stockwell C. W., "Determination of Feedback For Human Posture Control Without Physical Intervention," *Journal of Cybernetics*, 7:199-225, 1977.

[CRAI89] Craig, J., *Introduction to Robotics: Mechanics and Control*, Second Edition, Addison-Wesley Publishing Company, Inc., Menlo Park, California, 1989.

[DAVI93] Davidson, Sandra L., *An Experimental Comparison of CLOS and C++ Implementations of an Object-Oriented Graphical Simulation of Walking Robot Kinematics*, Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1993.

[DEVI96] DeVilliers, Edward Michael, *Implementing Voice Recognition and Natural Language Processing in the NPSNET Networked Virtual Environment* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1996.

[DURL95] Durlach, N. I. and Mavor, A. S., National Research Council, *Virtual Reality: Scientific and Technological Challenges*, National Academy Press, Washington, D.C., 1995, pp. 188-204, 306-317.

[FRAN69] Frank A. A. and McGhee R. B., "Some Considerations Relating to the Design of Autopilots for Legged Vehicles," *Journal of Terramechanics*, 1969, Vol. 6, No. 1, pp. 23 to 35.

[FREY96] Frey, William, III, *Application of Inertial Sensors and Flux-Gate Magnetometer to Real-Time Human Body Motion Capture,* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1996.

[GOET94] Goetz, John Robert, *Graphical Simulation of Articulated Rigid Body System Kinematics with Collision Detection*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

[GUBI74] Gubina, F., Hemami, H., McGhee, R. B., "On the Dynamic Stability of Biped Locomotion," *IEEE Transactions On Biomachanical Engineering*, BME-21, No. 2, 1974.

[KREY88] Kreyszig, Erwin, *Advanced Engineering Mathematics*, Sixth Edition, John Wiley & Sons, Toronto, 1988.

[KOOZ83] Koozekanani, S. H., Barin, K., McGhee, R. B., and Chang, H. T., "A Recursive Free-Body Approach to Computer Simulation of Human Postural Dynamics," *IEEE Transactions on Biomedical Engineering*, December 1983, Volume BME-30, Number 12, pp. 787-792.

[KOOZ80] Koozekanani, S. H., Stockwell, C. W., McGhee, R. B., Firoozmand, F., "On the Role of Dynamic Models in Quantitative Posturography," *IEEE Transactions On Biomedical Engineering*, BME-27, No. 10, 1980.

[KUO95] Kuo C. Benjanin, *Automatic Control Systems*, Seventh Edition, Prentice Hall Inc., A Simon & Schuster Company, Englewoodcliffs, New Jersey, 1995.

[MCGH74] McGhee, R. B., Pai, A. L., "An Approach to Computer Control for Legged Vehicles," *Journal of Terramechanics*, Vol. 11, No. 1, pp. 9 to 27, 1974.

[MCGH79] McGhee, R. B., "Computer Simulation of Human Movement", *CISM Courses and Lectures No. 263*, International Center for Mechanical Sciences, Springer-Verlag Wien-New York, 1980.

[MCGH86] McGhee, R. B., Nakano, E., Koyachi, N., Adachi, H., "*An Approach to Computer Coordination of Motion for Energy-Efficient Walking Machines*," Bulletin of Mechanical Engineering Laboratory, JAPAN, Number 43, 1986.

[MCMI94] McMillan, Scott, *Computational Dynamics for Robotic Systems on Land and Under Water*, Ph. D. Dissertation, Ohio State University, 1994.

[MCMI95] McMillan, S, Orin, D. E., and McGhee, R. B., "Efficient Dynamic Simulation of an Underwater Vehicle with a Robotic Manipulator," *IEEE Transactions On Systems, Man, and Cybernetics*, Volume 25, No. 8, 1995.

[GRAH96] Graham Paul, *ANSI Common Lisp*, Prentice Hall Inc., Englewoodcliffs, New Jersey, 1996.

[RAIB86] Raibert, Marc H., *Legged Robots That Balance*, The MIT Press, Cambridge, Massachusetts, London, England, 1986.

[TROY96] Troy, James J., *Dynamic Balance and Walking Control of Biped Mechanisms*, Ph. D. Dissertation, Iowa State University, 1995.

[WALD95] Waldrop, Marianne S., *Real-time Articulation of the Upper Body for Simulated Humans in Virtual Environments*, Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1995.

[WATT92] Watt, A. and Watt, M., *Advanced Animation and Rendering Techniques: Theory and Practice*, Addison-Wesley Publishing Company, Inc., New York, 1992, pp. 369-394.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. Robert B. McGhee, Professor 2
   Computer Science Department Code CS/MZ
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. Dr. Michael J. Zyda, Professor 2
   Computer Science Department Code CS/ZK
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. LT(jg) Mehmet Bediz 2
   60. sokak 144/28
   Emek
   Ankara 06510
   Turkey

7. Deniz Kuvvetleri Komutanligi 1
   Personel Tedarik ve Yetistirme Daire Baskanligi
   Bakanliklar, Ankara 06100
   Turkey

8. METU (ODTU) 1
   06531 Ankara
   Turkey

9. Bogazici University 1
80815 Bebek/Istanbul
Turkey

10. Bilkent University 1
Department of Computer Engineering and Information Science
06533 Bilkent/Ankara
Turkey